



Developer Guide

AWS HealthLake



AWS HealthLake: Developer Guide

Copyright © 2025 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What is AWS HealthLake?	1
Important notice	2
Features	2
Related services	3
Accessing	4
HIPAA	4
Pricing	4
Getting started	5
Concepts	5
authorization strategy	5
Integrated NLP	6
Integrated analytics	6
Setting up	6
Sign up for an AWS account	7
Create a user with administrative access	7
Configure an IAM user or role	9
Add a Data Lake Administrator user or role	11
Create S3 buckets	12
Create a data store	12
Set up import permissions	13
Set up export permissions	15
Install the AWS CLI	19
Tutorial	20
Managing data stores	21
Creating a data store	21
Getting data store properties	27
Listing data stores	30
Tagging data stores	33
Tagging a data store	34
Listing tags for a data store	36
Untagging a data store	39
Deleting a data store	41
Importing FHIR data	44
Starting an import job	46

Getting import job properties	50
Listing import jobs	52
Managing FHIR resources	57
Creating a resource	58
Reading a resource	62
Reading resource history	64
Reading version-specific history	67
Updating a resource	69
Conditional update	71
Configuring validation level for resource updates	72
Modifying a resource	73
Usage	73
Supported PATCH Operations	74
Request Headers	74
Examples	74
JSON Patch Path Syntax	76
Behavior	77
Error Handling	77
Caveats	77
Bundling resources	78
Bundle as independent entities	82
Conditional PUTs in Bundles	86
Bundle as a single entity	89
Configuring validation level for bundles	92
Deleting a resource	93
Conditional delete for FHIR	95
Idempotency and Concurrency	97
Idempotency Keys	97
ETag in AWS HealthLake	99
Searching FHIR resources	101
Searching with GET	101
GET search examples	104
Searching with POST	105
POST search examples	108
Search Consistency Levels	110
Consistency levels	110

Usage example	110
Best practices	111
Exporting FHIR data	113
Starting an export job	113
Getting export job properties	117
Listing export jobs	120
Code examples	124
Basics	124
Actions	125
Integrating	159
Natural language processing	159
NLP libraries	160
Using FHIR APIs	161
Search parameters	162
Example requests	165
SQL index and query	181
Getting started	182
Querying with SQL	185
Example queries	192
Monitoring	199
CloudTrail (API calls)	199
AWS HealthLake Information in CloudTrail	200
Understanding AWS HealthLake Log File Entries	201
CloudWatch (Metrics)	203
Viewing HealthLake metrics	206
Creating an alarm	206
EventBridge (Events)	207
HealthLake events sent to EventBridge	207
HealthLake event structure	208
Security	222
Data Protection	223
Encryption at rest	224
AWS owned KMS key	224
Customer managed KMS keys	224
Create a customer managed key	225
Required IAM permissions for using a customer managed KMS key	226

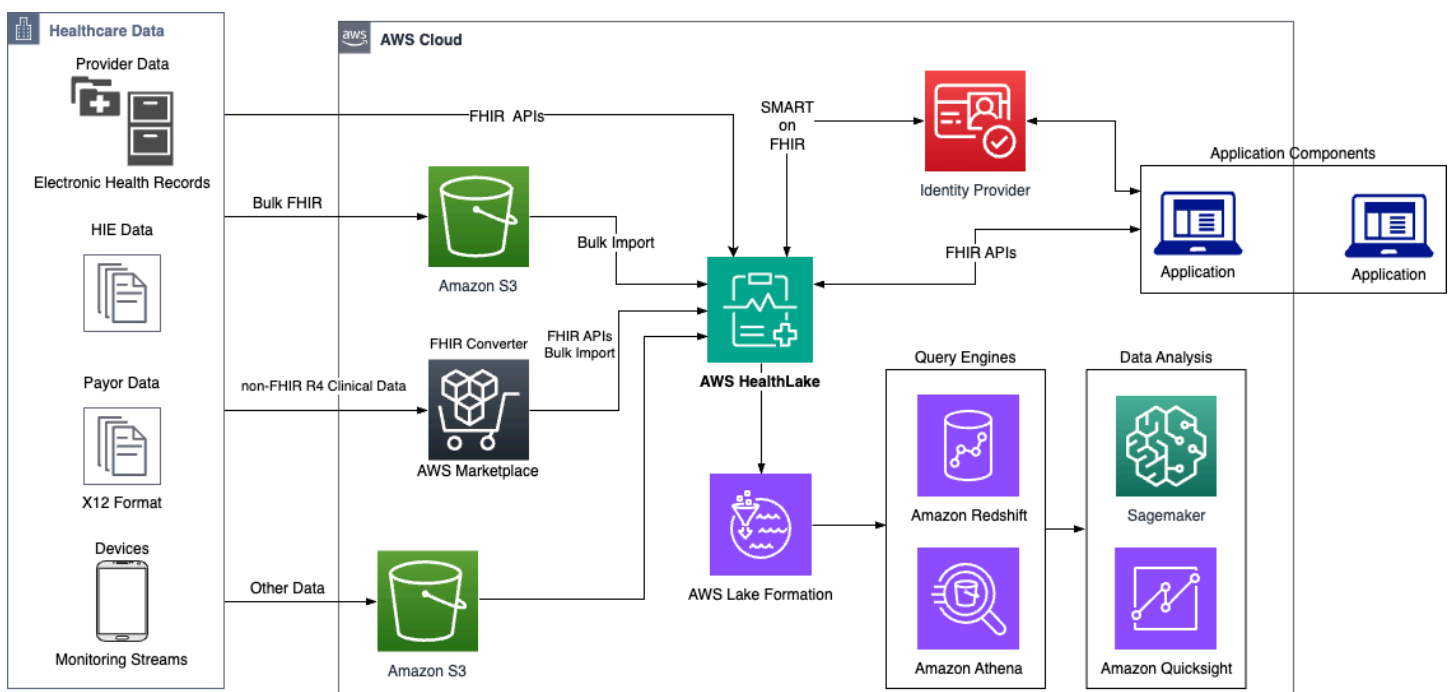
Encryption in transit	233
Identity and access management	233
Audience	234
Authenticating with identities	234
Managing access using policies	237
How AWS HealthLake works with IAM	240
Identity-based policy examples	246
AWS managed policies	250
Troubleshooting	254
Compliance validation	256
Infrastructure security	257
Infrastructure as code	257
HealthLake and AWS CloudFormation templates	258
Learn more about AWS CloudFormation	258
VPC endpoints	258
Considerations for HealthLake VPC endpoints	259
Creating an interface VPC endpoint for HealthLake;	259
Creating a VPC endpoint policy for HealthLake	259
Best practices	260
Resilience	261
Reference	262
SMART on FHIR	262
Getting started	263
Authentication	266
OAuth 2.0 scopes	267
Token validation	271
Fine-grained authorization	281
Discovery Document	282
Request example	283
FHIR R4	284
Capability Statement	285
Profile validations	286
Resource types	289
Search parameters	291
\$Operations	303
HealthLake	346

Endpoints and quotas	347
Preloaded data types	355
Sample projects	356
Troubleshooting	356
Working with AWS SDKs	364
Releases	366

What is AWS HealthLake?

AWS HealthLake is a HIPAA eligible service for storing, analyzing, and sharing health data in the cloud using the Fast Healthcare Interoperability Resources (FHIR) R4 specification. HealthLake use cases include:

- **Enterprise health data** – Manage and share FHIR R4 health data directly from AWS Cloud while preserving high performance and availability.
- **Healthcare interoperability** – Support customer conformance with 21st Century Cures Act for patient access through a fully managed FHIR data store.
- **Natural language processing (NLP)** – Utilize integrated NLP models to extract meaningful medical information from unstructured health data.
- **Multimodal analysis** – Combine HealthLake data with AWS HealthImaging data and AWS HealthOmics data to deliver insights for precision medicine.



Topics

- [Important notice](#)
- [Features of AWS HealthLake](#)
- [Related AWS services](#)

- [Accessing AWS HealthLake](#)
- [HIPAA eligibility and data security](#)
- [Pricing](#)

Important notice

AWS HealthLake is not a substitute for professional medical advice, diagnosis, or treatment, and is not intended to cure, treat, mitigate, prevent, or diagnose any disease or health condition. You are responsible for instituting human review as part of any use of AWS HealthLake, including in association with any third-party product intended to inform clinical decision-making. **AWS HealthLake should only be used in patient care or clinical scenarios after review by trained medical professionals applying sound medical judgment.**

Features of AWS HealthLake

AWS HealthLake provides the following features.

Import FHIR R4 health data

With the HealthLake native import action, you can easily migrate your FHIR data from an Amazon S3 bucket to an HealthLake data store, including clinical notes, lab reports, insurance claims, and more. HealthLake supports the FHIR R4 specification for health care data exchange. If needed, you can work with an [AWS HealthLake Partner](#) to convert your health data to FHIR R4 format.

Store health data in a secure, compliant, and auditable manner

A HealthLake data store helps index health data so it can be queried. The data store creates a complete view of each patient's medical history in chronological order and facilitates information exchange using the FHIR R4 specification. And it's always running to keep your index up to date, offering you the ability to search the information anytime using standard FHIR R4 interactions with durable primary storage and index scaling.

Leverage transactional FHIR server

Leverage FHIR APIs for standard resource validation, SMART on FHIR authorization, and Bulk data FHIR API export capabilities to support unifying and analyzing your data to reduce

operational costs and improve decision making. HealthLake supports customer conformity to the latest ONC and CMS regulatory standards including: HL7 FHIR R4 APIs, FHIR Bulk Data Access, US Core IG STU, HL7 SMART App Launch Framework IG, OAuth 2.0, and OpenID Connect.

Transform unstructured medical data using NLP

Integrated medical natural language processing (NLP) transforms all raw medical text data in a HealthLake data store to understand and extract meaningful information from unstructured healthcare data. With integrated medical NLP, you can automatically extract entities, entity relationships, entity traits, and protected health information (PHI) from your medical text. The NLP-extracted entities are stored as native FHIR R4 resources within a HealthLake data store and can be accessed through FHIR R4 APIs or Amazon Athena (SQL).

Related AWS services

AWS HealthLake features tight integration with other AWS services. A knowledge of the following services is useful to fully leverage HealthLake.

- [AWS Identity and Access Management](#) – Use IAM to securely manage identities and access to HealthLake resources.
- [Amazon Simple Storage Service](#) – Use Amazon S3 as a staging area to import DICOM data into HealthLake.
- [AWS CloudTrail](#) – Use CloudTrail to track HealthLake user activity and API usage.
- [Amazon CloudWatch](#) – Use CloudWatch to observe and monitor HealthLake resources.
- [AWS CloudFormation](#) – Use AWS CloudFormation to implement infrastructure as code (IaC) templates to create resources in HealthLake.
- [AWS PrivateLink](#) – Use Amazon VPC to establish connectivity between HealthLake and [Amazon Virtual Private Cloud](#) without exposing data to the internet.
- [Amazon EventBridge](#) – Use EventBridge to create scalable, event-driven applications by creating rules that route HealthLake events to targets.
- [AWS Lake Formation](#) – Use Lake Formation to centrally govern, secure, and share HealthLake data for analytics and machine learning.
- [Amazon Athena](#) – Use Athena to query HealthLake data with SQL to allow for deeper analysis.

Accessing AWS HealthLake

You can access AWS HealthLake using the AWS Management Console, AWS Command Line Interface and the AWS SDKs. This guide provides procedural instructions for the AWS Management Console and code examples for the AWS CLI and AWS SDKs.

AWS Command Line Interface (AWS CLI)

The AWS CLI provides commands for a broad set of AWS products, and is supported on Windows, Mac, and Linux. For more information, see the [AWS Command Line Interface User Guide](#).

AWS SDKs

AWS SDKs provide libraries, code examples, and other resources for software developers. These libraries provide basic functions that automate tasks such as cryptographically signing your requests, retrying requests, and handling error responses. For more information, see [Tools to Build on AWS](#).

AWS Management Console

The AWS Management Console provides a web-based user interface for managing HealthLake and its associated resources. If you've signed up for an AWS account, you can sign in to the [HealthLake Console](#).

HIPAA eligibility and data security

This is a HIPAA Eligible Service. For more information about AWS, U.S. Health Insurance Portability and Accountability Act of 1996 (HIPAA), and using AWS services to process, store, and transmit protected health information (PHI), see [HIPAA Overview](#).

Connections to HealthLake containing PHI and personally identifiable information (PII) must be encrypted. By default, all connections to HealthLake use HTTPS over TLS. HealthLake stores encrypted customer content and operates according to the [AWS Shared Responsibility Model](#).

Pricing

For HealthLake pricing information, see [AWS HealthLake pricing](#). To estimate costs, use the [HealthLake pricing calculator](#).

Getting started with AWS HealthLake

To start using AWS HealthLake, set up an AWS account and create an AWS Identity and Access Management user. To use the [AWS CLI](#) or the [AWS SDKs](#), you must install and configure them.

Note

The [Reference](#) chapter of this guide provides supporting content for SMART on FHIR, FHIR R4, and AWS HealthLake. For instance, you can find information about SMART on FHIR configuration, supported FHIR profile validations, and HealthLake endpoints.

After learning about HealthLake concepts and setting up, a short tutorial with code examples is available to help get you started.

Topics

- [AWS HealthLake concepts](#)
- [Setting up AWS HealthLake](#)
- [AWS HealthLake tutorial](#)

AWS HealthLake concepts

The following terminology and concepts are central to your understanding and use of AWS HealthLake.

Concepts

- [Data store authorization strategy](#)
- [Integrated NLP](#)
- [Integrated analytics](#)

Data store authorization strategy

A HealthLake data store is a repository of FHIR R4 health data that resides within a single AWS Region. HealthLake supports the following data store authorization strategies.

- **SigV4 authorization** — HealthLake authorizes FHIR API calls using [AWS Signature Version 4 \(SigV4\)](#) authorization.
- **SMART on FHIR authorization** — HealthLake authorizes FHIR API calls using [Substitutable Medical Applications and Reusable Technologies \(SMART\) on FHIR](#) authorization.

For more information, see [Creating a HealthLake data store](#).

Integrated NLP

AWS HealthLake integrates with HIPAA eligible natural language processing (NLP) libraries to extract meaningful health data from unstructured medical text. The NLP libraries identify medical entities like conditions, medications, dosages, tests, treatments, and procedures. They recognize relationships among the entities and link them to medical ontology libraries such as ICD-10-CM and RxNorm. For more information, see [Integrated natural language processing \(NLP\) for HealthLake](#).

Integrated analytics

AWS HealthLake goes beyond FHIR search and bundle APIs to provide integrated analytics for querying and analyzing large volumes of health data. During import, HealthLake automatically generates tables for SQL index and query. This enables you to gain actionable insights from complex healthcare data without requiring extensive data engineering work. For more information, see [Querying HealthLake data with Amazon Athena](#) and [AWS HealthLake sample projects](#).

Setting up AWS HealthLake

In this chapter, you use the AWS Management Console to set up the required permissions to start using AWS HealthLake and create a data store. To set up permissions to create a data store, you create an IAM user or role that is a data lake administrator and HealthLake administrator. You make this user a data lake administrator in AWS Lake Formation. The data lake administrator grants Lake Formation access to resources needed to use Amazon Athena to query a data store. After you create a HealthLake data store, you can set up permissions for importing and exporting files.

Topics

- [Sign up for an AWS account](#)
- [Create a user with administrative access](#)
- [Configure an IAM user or role to use HealthLake \(IAM Administrator\)](#)

- [Add a user or role as the Data Lake Administrator in Lake Formation \(IAM Administrator\)](#)
- [Create S3 buckets](#)
- [Create a data store](#)
- [Setting up permissions for import jobs](#)
- [Setting up permissions for export jobs](#)
- [Install the AWS CLI](#)

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open <https://portal.aws.amazon.com/billing/signup>.
2. Follow the online instructions.

Part of the sign-up procedure involves receiving a phone call or text message and entering a verification code on the phone keypad.

When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to a user, and use only the root user to perform [tasks that require root user access](#).

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to <https://aws.amazon.com/> and choosing **My Account**.

Create a user with administrative access

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the [AWS Management Console](#) as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

For help signing in by using root user, see [Signing in as the root user](#) in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

For instructions, see [Enable a virtual MFA device for your AWS account root user \(console\)](#) in the *IAM User Guide*.

Create a user with administrative access

1. Enable IAM Identity Center.

For instructions, see [Enabling AWS IAM Identity Center](#) in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to a user.

For a tutorial about using the IAM Identity Center directory as your identity source, see [Configure user access with the default IAM Identity Center directory](#) in the *AWS IAM Identity Center User Guide*.

Sign in as the user with administrative access

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

For help signing in using an IAM Identity Center user, see [Signing in to the AWS access portal](#) in the *AWS Sign-In User Guide*.

Assign access to additional users

1. In IAM Identity Center, create a permission set that follows the best practice of applying least-privilege permissions.

For instructions, see [Create a permission set](#) in the *AWS IAM Identity Center User Guide*.

2. Assign users to a group, and then assign single sign-on access to the group.

For instructions, see [Add groups](#) in the *AWS IAM Identity Center User Guide*.

Configure an IAM user or role to use HealthLake (IAM Administrator)

Persona: IAM Administrator

A user who can create IAM users and roles, and can add data lake administrators.

These steps in this topic must be carried out by an IAM administrator.

To connect your HealthLake data store to Athena, you need create an IAM user or role that is a data lake administrator and a HealthLake administrator. This new user or role grants access to resources found in a data store via AWS Lake Formation, and has the `AmazonHealthLakeFullAccess` AWS managed policy added to their user or role.

Important

An IAM user or role that is a data lake administrator *cannot* create new data lake administrators. To add additional data lake administrator you must use a IAM user or role which has been granted `AdministratorAccess` access.

To create an administrator

1. Add the **AmazonHealthlakeFullAccess** IAM AWS managed policy to a user or role in your organization.

If you're unfamiliar with creating an IAM user, see [Creating an IAM User](#) and [Overview of AWS IAM Policies](#) in the *IAM User Guide*.

2. Grant the IAM user or role access to AWS Lake Formation.
 - Add the following IAM AWS managed policy to a user or role in your organization:
AWSLakeFormationDataAdmin

Note

The `AWSLakeFormationDataAdmin` policy grants access to all AWS Lake Formation resources. We recommend that you always use the minimum permissions required to

accomplish your task. For more information, see [IAM Best Practices](#) in the *IAM User Guide*.

3. Add the following inline policy to the user or role. For more information, see [Inline policies](#) in the *IAM User Guide*.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::amzn-s3-demo-source-bucket/*",
        "arn:aws:s3:::amzn-s3-demo-logging-bucket/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "ram:GetResourceShareInvitations",
        "ram:AcceptResourceShareInvitation",
        "glue:CreateDatabase",
        "glue>DeleteDatabase"
      ],
      "Resource": "*"
    }
  ]
}
```

For more information on the AWSLakeFormationDataAdmin policy, see [Lake Formation Personas and IAM Permissions Reference](#) in the *AWS Lake Formation Developer Guide*.

Add a user or role as the Data Lake Administrator in Lake Formation (IAM Administrator)

Note

This step is required if you are integrating [SQL index and query](#).

Next, the IAM administrator must add the user or role created in the previous step as a data lake administrator in Lake Formation.

To add an IAM user or role as a data lake administrator

1. Open the AWS Lake Formation console: <https://console.aws.amazon.com/lakeformation/>

Note

If this is your first time visiting Lake Formation, a **Welcome to Lake Formation** dialog box appears asking you to define a Lake Formation administrator.

2. Assign the new user or role to be a AWS Lake Formation data lake administrator.
 - *Option 1:* If you received the **Welcome to Lake Formation** dialog box.
 1. Choose **Add other AWS users or roles**.
 2. Choose the **down arrow** (▼).
 3. Choose the HealthLake administrator you would like to also be Lake Formation administrators.
 4. Choose **Get started**.
 - *Option 2:* Use the **Navigation pane** (≡).
 1. Choose the **Navigation pane** (≡).
 2. Under **Permissions**, choose **Administrative roles and tasks**.
 3. In the **Data lake administrators** section, select **Choose administrators**.
 4. In the **Manage data lake administrators** dialog box, choose the **down arrow** (▼).
 5. Next, select or search for the HealthLake administrators users or roles who you also want to be Lake Formation administrators.

6. Choose **Save**.
3. Change the default security settings to be managed by Lake Formation. The HealthLake data store resources need to be managed by Lake Formation *not* IAM. To update, see [Change the default permission model](#) in the *AWS Lake Formation Developer Guide*.

Create S3 buckets

To import FHIR R4 data into AWS HealthLake, two Amazon S3 buckets are recommended. The Amazon S3 input bucket holds the FHIR data to be imported and HealthLake reads from this bucket. The Amazon S3 output bucket stores the processing results of the import job and HealthLake writes (logs) to this bucket.

Note

Due to AWS Identity and Access Management (IAM) policy, your Amazon S3 bucket names must be unique. For more information, see [Bucket naming rules](#) in the *Amazon Simple Storage Service User Guide*.

For the purpose of this guide, we specify the following Amazon S3 input and output buckets when setting up [import permissions](#) later in this section.

- Input bucket: `arn:aws:s3:::amzn-s3-demo-source-bucket`
- Output bucket: `arn:aws:s3:::amzn-s3-demo-logging-bucket`

For additional information, see [Creating a bucket](#) in the *Amazon S3 User Guide*.

Create a data store

A HealthLake data store is a repository of FHIR R4 data that resides within a single AWS Region. An AWS account can have zero or many data stores. HealthLake supports two data store [authorization strategies](#).

Important

Before you create a HealthLake data store, review the [Service control policies \(SCPs\)](#) in your AWS Organization that might restrict the creation or management of HealthLake

resources. SCPs can prevent the successful creation of HealthLake data stores, even if your IAM permissions are set up correctly.

A `datastoreID` is generated when you create a HealthLake data store. You must use the `datastoreID` when setting up [import permissions](#) later in this section.

To create a HealthLake data store, see [Creating a HealthLake data store](#).

Setting up permissions for import jobs

Before you import files into a data store, you must grant HealthLake permission to access your input and output buckets in Amazon S3. To grant HealthLake access, you create an IAM service role for HealthLake, add a trust policy to the role to grant HealthLake assume role permissions, and attach a permissions policy to role that grants it to access to your Amazon S3 buckets.

When you create an import job, you specify the Amazon Resource Name (ARN) of this role for the `DataAccessRoleArn`. For more information about IAM roles and trust policies, see [IAM Roles](#).

After you set up permission, you are ready to import files into your data store with an import job. For more information, see [Starting a FHIR import job](#).

To set up import permissions

1. If haven't already, create a destination Amazon S3 bucket for output log files. The Amazon S3 bucket must be in the same AWS Region as the service, and Block Public Access must be turned on for all options. To learn more, see [Using Amazon S3 block public access](#). An Amazon-owned or customer-owned KMS key must also be used for encryption. To learn more about using KMS keys, see [Amazon Key Management Service](#).
2. Create a data access service role for HealthLake and give the HealthLake service permission to assume it with the following trust policy. HealthLake uses this to write the output Amazon S3 bucket.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```

        "Principal": {
            "Service": [
                "healthlake.amazonaws.com"
            ]
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "accountID"
            },
            "ArnEquals": {
                "aws:SourceArn": "arn:aws:healthlake:us-
west-2:111122223333:datastore/fhir/datastoreID"
            }
        }
    }
}

```

3. Add a permissions policy to the data access role that allows it to access the Amazon S3 bucket. Replace `amzn-s3-demo-bucket` with your bucket's name.

JSON

```

{
    "Version": "2012-10-17",
    "Statement": [{
        "Action": [
            "s3:ListBucket",
            "s3:GetBucketPublicAccessBlock",
            "s3:GetEncryptionConfiguration"
        ],
        "Resource": [
            "arn:aws:s3:::amzn-s3-demo-source-bucket"
        ],
        "Effect": "Allow"
    },
    {
        "Action": [
            "s3:PutObject"
        ],
        "Resource": [
            "arn:aws:s3:::amzn-s3-demo-logging-bucket/*"
        ]
    }
]
}

```

```
    ],
    "Effect": "Allow"
  },
  {
    "Action": [
      "kms:DescribeKey",
      "kms:GenerateDataKey*"
    ],
    "Resource": [
      "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-b56c-4216-a250-f4c43ef46e83"
    ],
    "Effect": "Allow"
  }
]
```

Setting up permissions for export jobs

Before you export files from a data store, you must grant HealthLake permission to access your output bucket in Amazon S3. To grant HealthLake access, you create an IAM service role for HealthLake, add a trust policy to the role to grant HealthLake assume role permissions, and attach a permissions policy to role that grants it to access to your Amazon S3 bucket.

If you already created a role for HealthLake, you can reuse it and grant it the additional permissions for your export Amazon S3 bucket listed in this topic. To learn more about IAM roles and trust policies, see [IAM Policies and Permissions](#).

Important

HealthLake supports both [native SDK export requests](#) and the [FHIR R4 \\$export](#) operation. Separate IAM actions must be provided depending on which export API you decide to use. This allows you to handle allow and deny permissions separately. If you want to restrict both HealthLake SDK and FHIR REST API exports, you must apply deny permissions to the separate IAM actions. IAM user permission changes are not required if you give users full access to HealthLake.

Using AWS CLI and AWS SDKs:

The following native HealthLake actions are available for exporting data from a data store using the AWS CLI and AWS SDKs:

- StartFHIRExportJob
- DescribeFHIRExportJob
- ListFHIRExportJobs

Using FHIR APIs:

The following IAM actions are available for exporting data from a HealthLake data store and for cancelling (deleting) an export job using the FHIR \$export operation:

POST:

- StartFHIRExportJobWithPost

GET:

- StartFHIRExportJobWithGet
- DescribeFHIRExportJobWithGet
- GetExportedFile

DELETE:

- CancelFHIRExportJobWithDelete

The user or role that sets up permissions must have permission to create roles, create policies, and attach policies to roles. The following IAM policy grants these permissions.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "iam:CreateRole",
        "iam:CreatePolicy",
        "iam:AttachRolePolicy"
      ],
    },
  ],
}
```

```
        "Effect": "Allow",
        "Resource": "*"
    },
    {
        "Action": "iam:PassRole",
        "Effect": "Allow",
        "Resource": "*",
        "Condition": {
            "StringEquals": {
                "iam:PassedToService": "healthlake.amazonaws.com"
            }
        }
    }
]
```

To set up export permissions

1. If haven't already, create a destination Amazon S3 bucket for the data you will export from your data store. The Amazon S3 bucket must be in the same AWS Region as the service, and Block Public Access must be turned on for all options. To learn more, see [Using Amazon S3 block public access](#). An Amazon-owned or customer-owned KMS key must also be used for encryption. To learn more about using KMS keys, see [Amazon Key Management Service](#).
2. If you haven't already, create a data access service role for HealthLake and give the HealthLake service permission to assume it with the following trust policy. HealthLake uses this to write the output Amazon S3 bucket. If you already created one in [Setting up permissions for import jobs](#), you can reuse it and grant it permissions for your Amazon S3 bucket in the next step.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "healthlake.amazonaws.com"
        ]
      }
    }
  ],
}
```



```

        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "accountID"
            },
            "ArnEquals": {
                "aws:SourceArn": "arn:aws:healthlake:us-
west-2:111122223333:datastore/fhir/data store ID"
            }
        }
    }
}

```

3. Add a permissions policy to the data access role that allows it to access your output Amazon S3 bucket. Replace `amzn-s3-demo-bucket` with your bucket's name.

JSON

```

{
    "Version": "2012-10-17",
    "Statement": [{
        "Action": [
            "s3:ListBucket",
            "s3:GetBucketPublicAccessBlock",
            "s3:GetEncryptionConfiguration"
        ],
        "Resource": [
            "arn:aws:s3:::amzn-s3-demo-source-bucket"
        ],
        "Effect": "Allow"
    },
    {
        "Action": [
            "s3:PutObject"
        ],
        "Resource": [
            "arn:aws:s3:::amzn-s3-demo-logging-bucket/*"
        ],
        "Effect": "Allow"
    },
    {
        "Action": [

```

```
        "kms:DescribeKey",
        "kms:GenerateDataKey*"
    ],
    "Resource": [
        "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-b56c-4216-a250-
f4c43ef46e83"
    ],
    "Effect": "Allow"
  }]
}
```

Install the AWS CLI

The AWS CLI is required to describe and list HealthLake import and export job properties. You can also request this information using HealthLake SDKs.

To set up the AWS CLI

1. Download and configure the AWS CLI. For instructions, see the following topics in the *AWS Command Line Interface User Guide*.
 - [Installing or updating the latest version of the AWS CLI](#)
 - [Getting started with the AWS CLI](#)
2. In the AWS CLI config file, add a named profile for the administrator. You use this profile when running the AWS CLI commands. Under the security principle of least privilege, we recommend you create a separate IAM role with privileges specific to the tasks being performed. For more information about named profiles, see [Configuration and credential file settings](#) in the *AWS Command Line Interface User Guide*.

```
[default]
aws_access_key_id = default access key ID
aws_secret_access_key = default secret access key
region = region
```

3. Verify the setup using the following help command.

```
aws healthlake help
```

If the AWS CLI is configured correctly, you see a brief description of AWS HealthLake and a list of available commands.

AWS HealthLake tutorial

Objective

In this tutorial, you will import FHIR R4 data into a HealthLake data store using native HealthLake actions. Next, you will manage (create, read, update, delete) a FHIR resource using FHIR RESTful APIs. To conclude the tutorial, you will export FHIR data using native HealthLake actions.

Prerequisites

All procedures listed in [Setting up](#) are required to complete this tutorial.

Tutorial steps

1. [Start FHIR import job](#)
2. [Get FHIR import job properties](#)
3. [Create FHIR resource](#)
4. [Read FHIR resource](#)
5. [Update FHIR resource](#)
6. [Delete FHIR resource](#)
7. [Export FHIR data](#)
8. [Delete data store](#)

Managing data stores with AWS HealthLake

With AWS HealthLake, you create and manage data stores for FHIR R4 resources. When you create a HealthLake data store, a FHIR data repository is made available via a RESTful API [endpoint](#). You can choose to import (preload) Synthea open source FHIR R4 health data into your data store when you create it. For more information, see [Preloaded data types](#).

Important

HealthLake supports two types of FHIR data store authorization strategies, AWS SigV4 or SMART on FHIR. You must choose one of the authorization strategies prior to creating a HealthLake FHIR data store. For more information, see [Data store authorization strategy](#).

To find the FHIR-related capabilities (behaviors) of an active HealthLake data store, retrieve its [Capability Statement](#).

The following topics describe how to use HealthLake cloud native actions to create, describe, list, tag, and delete FHIR data stores using the AWS CLI, AWS SDKs, and AWS Management Console.

Topics

- [Creating a HealthLake data store](#)
- [Getting HealthLake data store properties](#)
- [Listing HealthLake data stores](#)
- [Tagging HealthLake data stores](#)
- [Deleting a HealthLake data store](#)

Creating a HealthLake data store

Use `CreateFHIRDatastore` to create an AWS HealthLake data store conformant to the FHIR R4 specification. HealthLake data stores are used for importing, managing, searching, and exporting FHIR data. You can choose to import (preload) Synthea open source FHIR R4 health data into your data store when you create it. For more information, see [Preloaded data types](#).

Important

HealthLake supports two types of FHIR data store authorization strategies, AWS SigV4 or SMART on FHIR. You must choose one of the authorization strategies prior to creating a HealthLake FHIR data store. For more information, see [Data store authorization strategy](#).

When you create a HealthLake data store, a FHIR data repository is made available via a RESTful API [endpoint](#). After you've created your HealthLake data store, you can request its [Capability Statement](#) to find all associated FHIR-related capabilities (behaviors).

The following menus provide examples for the AWS CLI and AWS SDKs and a procedure for the AWS Management Console. For more information, see [CreateFHIRDatastore](#) in the *AWS HealthLake API Reference*.

To create a HealthLake data store

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

Example 1: Create a SigV4-enabled HealthLake data store

The following `create-fhir-datastore` example demonstrates how to create a new data store in AWS HealthLake.

```
aws healthlake create-fhir-datastore \
  --datastore-type-version R4 \
  --datastore-name "FhirTestDatastore"
```

Output:

```
{
  "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
(Data store ID)/r4/",
  "DatastoreArn": "arn:aws:healthlake:us-east-1:(AWS Account ID):datastore/
(Data store ID)",
  "DatastoreStatus": "CREATING",
```

```
"DatastoreId": "(Data store ID)"
}
```

Example 2: Create a SMART on FHIR-enabled HealthLake data store

The following `create-fhir-datastore` example demonstrates how to create a new SMART on FHIR-enabled data store in AWS HealthLake.

```
aws healthlake create-fhir-datastore \
  --datastore-name "your-data-store-name" \
  --datastore-type-version R4 \
  --preload-data-config PreloadDataType="SYNTHETA" \
  --sse-configuration '{ "KmsEncryptionConfig": { "CmkType":  
"CUSTOMER_MANAGED_KMS_KEY", "KmsKeyId": "arn:aws:kms:us-east-1:your-account-  
id:key/your-key-id" } }' \
  --identity-provider-configuration file://  
identity_provider_configuration.json
```

Contents of `identity_provider_configuration.json`:

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR_V1",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-  
lambda-name",
  "Metadata": "{ \"issuer\": \"https://ehr.example.com\", \"jwks_uri\":  
\"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint  
\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://  
ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\":  
[\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credential  
\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/  
register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"],  
\"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://  
ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://  
ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://  
ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"],  
\"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\"] }"
```

Output:

```
{
```

```

    "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
(Data store ID)/r4/",
    "DatastoreArn": "arn:aws:healthlake:us-east-1:(AWS Account ID):datastore/
(Data store ID)",
    "DatastoreStatus": "CREATING",
    "DatastoreId": "(Data store ID)"
}

```

- For API details, see [CreateFHIRDatastore](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def create_fhir_datastore(
    self,
    datastore_name: str,
    sse_configuration: dict[str, any] = None,
    identity_provider_configuration: dict[str, any] = None,
) -> dict[str, str]:
    """
    Creates a new HealthLake data store.
    When creating a SMART on FHIR data store, the following parameters are
    required:
    - sse_configuration: The server-side encryption configuration for a SMART
    on FHIR-enabled data store.
    - identity_provider_configuration: The identity provider configuration
    for a SMART on FHIR-enabled data store.

```

```

        :param datastore_name: The name of the data store.
        :param sse_configuration: The server-side encryption configuration for a
SMART on FHIR-enabled data store.
        :param identity_provider_configuration: The identity provider
configuration for a SMART on FHIR-enabled data store.
        :return: A dictionary containing the data store information.
        """
        try:
            parameters = {"DatastoreName": datastore_name,
"DatastoreTypeVersion": "R4"}
            if (
                sse_configuration is not None
                and identity_provider_configuration is not None
            ):
                # Creating a SMART on FHIR-enabled data store
                parameters["SseConfiguration"] = sse_configuration
                parameters[
                    "IdentityProviderConfiguration"
                ] = identity_provider_configuration

            response =
self.health_lake_client.create_fhir_datastore(**parameters)
            return response
        except ClientError as err:
            logger.exception(
                "Couldn't create data store %s. Here's why %s",
                datastore_name,
                err.response["Error"]["Message"],
            )
            raise

```

The following code shows an example of parameters for a SMART on FHIR-enabled HealthLake data store.

```

sse_configuration = {
    "KmsEncryptionConfig": {"CmkType": "AWS_OWNED_KMS_KEY"}
}
# TODO: Update the metadata to match your environment.
metadata = {
    "issuer": "https://ehr.example.com",
    "jwks_uri": "https://ehr.example.com/.well-known/jwks.json",

```



```

        "authorization_endpoint": "https://ehr.example.com/auth/
authorize",
        "token_endpoint": "https://ehr.token.com/auth/token",
        "token_endpoint_auth_methods_supported": [
            "client_secret_basic",
            "foo",
        ],
        "grant_types_supported": ["client_credential", "foo"],
        "registration_endpoint": "https://ehr.example.com/auth/register",
        "scopes_supported": ["openId", "profile", "launch"],
        "response_types_supported": ["code"],
        "management_endpoint": "https://ehr.example.com/user/manage",
        "introspection_endpoint": "https://ehr.example.com/user/
introspect",
        "revocation_endpoint": "https://ehr.example.com/user/revoke",
        "code_challenge_methods_supported": ["S256"],
        "capabilities": [
            "launch-ehr",
            "sso-openid-connect",
            "client-public",
        ],
    }
    # TODO: Update the IdpLambdaArn.
    identity_provider_configuration = {
        "AuthorizationStrategy": "SMART_ON_FHIR_V1",
        "FineGrainedAuthorizationEnabled": True,
        "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-
id:function:your-lambda-name",
        "Metadata": json.dumps(metadata),
    }
    data_store = self.create_fhir_datastore(
        datastore_name, sse_configuration,
        identity_provider_configuration
    )

```

- For API details, see [CreateFHIRDatastore](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

Note

The following procedure creates a HealthLake data store with [AWS SigV4](#) authorization. The HealthLake Console does not support the creation of a SMART on FHIR data store.

To create a HealthLake data store with AWS SigV4 authorization

1. Sign in to the [Create data store](#) page on the HealthLake Console.
2. Choose **Create Data Store**.
3. In the **Data Store settings** section, for **Data Store name**, specify a name.
4. (Optional) In the **Data Store settings** section, for **Preload sample data**, select the check box to preload Synthea data. Synthea data is an open-source sample dataset. For more information, see [Synthea preloaded data types for HealthLake](#).
5. In the **Data Store encryption** section, choose either **Use AWS owned key (default)** or **Choose a different AWS KMS key (advanced)**.
6. In the **Tags - optional** section, you can add tags to your data store. To learn more about tagging your data store, see [Tagging HealthLake data stores](#).
7. Choose **Create Data Store**.

The status of your data store is available on the **Data stores** page.

Getting HealthLake data store properties

Use `DescribeFHIRDatastore` to get properties for an AWS HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [DescribeFHIRDatastore](#) in the *AWS HealthLake API Reference*.

To get properties for a HealthLake data store

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To describe a FHIR data store

The following `describe-fhir-datastore` example demonstrates how to find the properties of a data store in AWS HealthLake.

```
aws healthlake describe-fhir-datastore \
  --datastore-id "1f2f459836ac6c513ce899f9e4f66a59"
```

Output:

```
{
  "DatastoreProperties": {
    "PreloadDataConfig": {
      "PreloadDataType": "SYNTHEA"
    },
    "SseConfiguration": {
      "KmsEncryptionConfig": {
        "CmkType": "CUSTOMER_MANAGED_KMS_KEY",
        "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
      }
    },
    "DatastoreName": "Demo",
    "DatastoreArn": "arn:aws:healthlake:us-east-1:<AWS Account ID>:datastore/<Data store ID>",
    "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/<Data store ID>/r4/",
    "DatastoreStatus": "ACTIVE",
    "DatastoreTypeVersion": "R4",
    "CreatedAt": 1603761064.881,
    "DatastoreId": "<Data store ID>",
    "IdentityProviderConfiguration": {
      "AuthorizationStrategy": "AWS_AUTH",

```

```

        "FineGrainedAuthorizationEnabled": false
    }
}
}

```

- For API details, see [DescribeFHIRDatastore](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def describe_fhir_datastore(self, datastore_id: str) -> dict[str, any]:
    """
    Describes a HealthLake data store.
    :param datastore_id: The data store ID.
    :return: The data store description.
    """
    try:
        response = self.health_lake_client.describe_fhir_datastore(
            DatastoreId=datastore_id
        )
        return response["DatastoreProperties"]
    except ClientError as err:
        logger.exception(
            "Couldn't describe data store with ID %s. Here's why %s",
            datastore_id,
            err.response["Error"]["Message"],
        )
        raise

```

- For API details, see [DescribeFHIRDatastore](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

1. Sign in to the [Data stores](#) page on the HealthLake Console.
2. Choose a data store.

The **Data Store details** page opens and all HealthLake data store properties are available.

Listing HealthLake data stores

Use `ListFHIRDatastores` to list all HealthLake data stores in a user's account, regardless of data store status. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [ListFHIRDatastores](#) in the *AWS HealthLake API Reference*.

To list all HealthLake data stores

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To list FHIR data stores

The following `list-fhir-datastores` example shows to how to use the command and how users can filter results based on data store status in AWS HealthLake.

```
aws healthlake list-fhir-datastores \
  --filter DatastoreStatus=ACTIVE
```

Output:

```
{
  "DatastorePropertiesList": [
    {
      "PreloadDataConfig": {
        "PreloadDataType": "SYNTHEA"
      },
      "SseConfiguration": {
        "KmsEncryptionConfig": {
          "CmkType": "CUSTOMER_MANAGED_KMS_KEY",
          "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
        }
      },
      "DatastoreName": "Demo",
      "DatastoreArn": "arn:aws:healthlake:us-east-1:<AWS Account ID>:datastore/<Data store ID>",
      "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/<Data store ID>/r4/",
      "DatastoreStatus": "ACTIVE",
      "DatastoreTypeVersion": "R4",
      "CreatedAt": 1603761064.881,
      "DatastoreId": "<Data store ID>",
      "IdentityProviderConfiguration": {
        "AuthorizationStrategy": "AWS_AUTH",
        "FineGrainedAuthorizationEnabled": false
      }
    }
  ]
}
```

```
]
}
```

- For API details, see [ListFHIRDatastores](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def list_fhir_datastores(self) -> list[dict[str, any]]:
    """
    Lists all HealthLake data stores.
    :return: A list of data store descriptions.
    """
    try:
        next_token = None
        datastores = []

        # Loop through paginated results.
        while True:
            parameters = {}
            if next_token is not None:
                parameters["NextToken"] = next_token
            response =
self.health_lake_client.list_fhir_datastores(**parameters)
            datastores.extend(response["DatastorePropertiesList"])
            if "NextToken" in response:
                next_token = response["NextToken"]
            else:
```

```
        break

    return datastores
except ClientError as err:
    logger.exception(
        "Couldn't list data stores. Here's why %s", err.response["Error"]
["Message"]
    )
    raise
```

- For API details, see [ListFHIRDatastores](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

- Sign in to the [Data stores](#) page on the HealthLake Console.

All HealthLake data stores are listed under the **Data stores** section.

Tagging HealthLake data stores

You can assign metadata to HealthLake data stores in the form of tags. Each tag is a label consisting of a user-defined key and value. Tags help you manage, identify, organize, search for, and filter data stores.

Important

Do not store protected health information (PHI), personally identifiable information (PII), or other confidential or sensitive information in tags. Tags are not intended to be used for private or sensitive data.

The following topics describe how to use HealthLake tagging operations using the AWS Management Console, AWS CLI, and AWS SDKs. For more information, see [Tagging your AWS resources](#) in the *AWS General Reference Guide*.

Topics

- [Tagging a HealthLake data store](#)
- [Listing tags for a HealthLake data store](#)
- [Untagging a HealthLake data store](#)

Tagging a HealthLake data store

Use TagResource to tag a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [TagResource](#) in the *AWS HealthLake API Reference*.

To tag a HealthLake data store

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs**CLI****AWS CLI****To add a tag to data store**

The following tag-resource example shows how to add a tag to a data store.

```
aws healthlake tag-resource \  
  --resource-arn "arn:aws:healthlake:us-east-1:123456789012:datastore/  
  fhix/0725c83f4307f263e16fd56b6d8ebdbe" \  
  --tag-key "Key" --tag-value "Value"
```

```
--tags '[{"Key": "key1", "Value": "value1"}]'
```

This command produces no output.

- For API details, see [TagResource](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def tag_resource(self, resource_arn: str, tags: list[dict[str, str]]) ->
None:
    """
    Tags a HealthLake resource.
    :param resource_arn: The resource ARN.
    :param tags: The tags to add to the resource.
    """
    try:
        self.health_lake_client.tag_resource(ResourceARN=resource_arn,
        Tags=tags)
    except ClientError as err:
        logger.exception(
            "Couldn't tag resource %s. Here's why %s",
            resource_arn,
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [TagResource](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

1. Sign in to the [Data stores](#) page on the HealthLake Console.
2. Choose a data store.

The **Data store details** page opens.

3. Under the **Tags** section, choose **Manage tags**.

The **Manage tags** page opens.

4. Choose **Add new tag**.
5. Enter a **Key** and **Value** (optional).
6. Choose **Save**.

Listing tags for a HealthLake data store

Use `ListTagsForResource` to list tags for a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [ListTagsForResource](#) in the *AWS HealthLake API Reference*.

To list tags for a HealthLake data store

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To list tags for a data store

The following `list-tags-for-resource` example lists the tags associated with the specified data store.:

```
aws healthlake list-tags-for-resource \
    --resource-arn "arn:aws:healthlake:us-east-1:123456789012:datastore/
    fh1r/0725c83f4307f263e16fd56b6d8ebdbe"
```

Output:

```
{
  "tags": {
    "key": "value",
    "key1": "value1"
  }
}
```

- For API details, see [ListTagsForResource](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)
```

```
def list_tags_for_resource(self, resource_arn: str) -> dict[str, str]:
    """
    Lists the tags for a HealthLake resource.
    :param resource_arn: The resource ARN.
    :return: The tags for the resource.
    """
    try:
        response = self.health_lake_client.list_tags_for_resource(
            ResourceARN=resource_arn
        )
        return response["Tags"]
    except ClientError as err:
        logger.exception(
            "Couldn't list tags for resource %s. Here's why %s",
            resource_arn,
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [ListTagsForResource](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

1. Sign in to the [Data stores](#) page on the HealthLake Console.
2. Choose a data store.

The **Data store details** page opens. Under the **Tags** section, all data store tags are listed.

Untagging a HealthLake data store

Use `UntagResource` to remove a tag from a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [UntagResource](#) in the *AWS HealthLake API Reference*.

To untag a HealthLake data store

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To remove tags from a data store.

The following `untag-resource` example shows how to remove tags from a data store.

```
aws healthlake untag-resource \
  --resource-arn "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
b91723d65c6fdeb1d26543a49d2ed1fa" \
  --tag-keys '["key1"]'
```

This command produces no output.

- For API details, see [UntagResource](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.
```

```
        :return: An instance of HealthLakeWrapper initialized with the default
        HealthLake client.
        """
        health_lake_client = boto3.client("healthlake")
        return cls(health_lake_client)

def untag_resource(self, resource_arn: str, tag_keys: list[str]) -> None:
    """
    Untags a HealthLake resource.
    :param resource_arn: The resource ARN.
    :param tag_keys: The tag keys to remove from the resource.
    """
    try:
        self.health_lake_client.untag_resource(
            ResourceARN=resource_arn, TagKeys=tag_keys
        )
    except ClientError as err:
        logger.exception(
            "Couldn't untag resource %s. Here's why %s",
            resource_arn,
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [UntagResource](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

1. Sign in to the [Data stores](#) page on the HealthLake Console.
2. Choose a data store.

The **Data store details** page opens.

3. Under the **Tags** section, choose **Manage tags**.

The **Manage tags** page opens.

4. Choose **Remove** next to the tag you want to remove.
5. Choose **Save**.

Deleting a HealthLake data store

Use `DeleteFHIRDatastore` to delete a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [DeleteFHIRDatastore](#) in the *AWS HealthLake API Reference*.

To delete a HealthLake data store

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To delete a FHIR data store

The following `delete-fhir-datastore` example demonstrates how to delete a data store and all of its contents in AWS HealthLake.

```
aws healthlake delete-fhir-datastore \  
  --datastore-id (Data store ID)
```

Output:

```
{
```



```

    "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
(Data store ID)/r4/",
    "DatastoreArn": "arn:aws:healthlake:us-east-1:(AWS Account ID):datastore/
(Data store ID)",
    "DatastoreStatus": "DELETING",
    "DatastoreId": "(Data store ID)"
}

```

- For API details, see [DeleteFHIRDatastore](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def delete_fhir_datastore(self, datastore_id: str) -> None:
    """
    Deletes a HealthLake data store.
    :param datastore_id: The data store ID.
    """
    try:
        self.health_lake_client.delete_fhir_datastore(DatastoreId=datastore_id)
    except ClientError as err:
        logger.exception(
            "Couldn't delete data store with ID %s. Here's why %s",
            datastore_id,
            err.response["Error"]["Message"],
        )
    raise

```

- For API details, see [DeleteFHIRDatastore](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

1. Sign in to the [Data stores](#) page on the HealthLake Console.
2. Choose a data store.

The **Data store details** page opens.

3. Choose **Delete**.

The **Delete data store** page opens.

4. To confirm data store deletion, enter the data store name in the text input field.
5. Choose **Delete**.

Importing FHIR data with AWS HealthLake

After creating a HealthLake data store, the next step is to import files from an Amazon Simple Storage Service (S3) bucket. You can start a FHIR import job using the AWS Management Console, AWS CLI, or AWS SDKs. Use native AWS HealthLake actions to start, describe, and list FHIR import jobs.

Important

HealthLake supports the [FHIR R4 specification](#) for health care data exchange. If needed, you can work with an [AWS HealthLake Partner](#) to convert your health data to FHIR R4 format prior to import.

When starting a FHIR import job, you specify an Amazon S3 bucket input location, an Amazon S3 bucket output location (for job processing results), an IAM role that grants HealthLake access to your Amazon S3 buckets, and a customer owned or AWS owned AWS Key Management Service key. For more information, see [Setting up permissions for import jobs](#).

Note

You can queue import jobs. The asynchronous import jobs are processed in a FIFO (First In First Out) manner. You can queue jobs the same way you start import jobs. If one is in progress, it will simply queue up. You can create, read, update, or delete FHIR resources while an import job is in progress.

HealthLake generates a `manifest.json` file for each FHIR import job. The file describes both the successes and failures of a FHIR import job. HealthLake outputs the `manifest.json` file to the Amazon S3 bucket specified when starting a FHIR import job. Log files are organized into two folders, named `SUCCESS` and `FAILURE`. Use the `manifest.json` file as the first step in troubleshooting a failed import job, as it provides details on each file.

```
{
  "inputDataConfig": {
    "s3Uri": "s3://amzn-s3-demo-source-bucket/healthlake-input/invalidInput/"
  },
}
```

```

    "outputDataConfig": {
      "s3Uri": "s3://amzn-s3-demo-logging-bucket/32839038a2f47f17c2fe0f53f0c3a0ba-
FHIR_IMPORT-19dd7bb7bcc8ee12a09bf6d322744a3d/",
      "encryptionKeyID": "arn:aws:kms:us-west-2:123456789012:key/fbbbf3e3-20b3-42a5-
a99d-c48c655ed545"
    },
    "successOutput": {
      "successOutputS3Uri": "s3://amzn-s3-demo-logging-
bucket/32839038a2f47f17c2fe0f53f0c3a0ba-FHIR_IMPORT-19dd7bb7bcc8ee12a09bf6d322744a3d/
SUCCESS/"
    },
    "failureOutput": {
      "failureOutputS3Uri": "s3://amzn-s3-demo-logging-
bucket/32839038a2f47f17c2fe0f53f0c3a0ba-FHIR_IMPORT-19dd7bb7bcc8ee12a09bf6d322744a3d/
FAILURE/"
    },
    "numberOfScannedFiles": 1,
    "numberOfFilesImported": 1,
    "sizeOfScannedFilesInMB": 0.023627,
    "sizeOfDataImportedSuccessfullyInMB": 0.011232,
    "numberOfResourcesScanned": 9,
    "numberOfResourcesImportedSuccessfully": 4,
    "numberOfResourcesWithCustomerError": 5,
    "numberOfResourcesWithServerError": 0
  }
}

```

Configuring validation level for imports

When starting a FHIR import job, you can optionally specify a `ValidationLevel` to apply to each resource. AWS HealthLake currently supports the following validation levels:

- **strict**: Resources are validated according to the profile element of the resource, or the R4 specification if no profile is present. This is the default validation level for AWS HealthLake.
- **structure-only**: Resources are validated against R4, ignoring any referenced profiles.
- **minimal**: Resources are validated minimally, ignoring certain R4 rules. Resources that fail structure checks required for search/analytics will be updated to include a warning for audit.

When importing using the `minimal` validation level, additional log files may be generated in a folder named `SUCCESS_WITH_SEARCH_VALIDATION_FAILURES`. Resources within the log files of this folder were ingested into your datastore despite failing search related validation checks. This

implies certain aspects of your FHIR resource were invalid according to FHIR, and malformed fields may not be searchable. These resources will have an extension appended to them describing said failure.

Topics

- [Starting a FHIR import job](#)
- [Getting FHIR import job properties](#)
- [Listing FHIR import jobs](#)

Starting a FHIR import job

Use `StartFHIRImportJob` to start a FHIR import job into a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [StartFHIRImportJob](#) in the *AWS HealthLake API Reference*.

Important

HealthLake supports the [FHIR R4 specification](#) for health care data exchange. If needed, you can work with an [AWS HealthLake Partner](#) to convert your health data to FHIR R4 format prior to import.

To start a FHIR import job

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To start a FHIR import job

The following `start-fhir-import-job` example shows how to start a FHIR import job using AWS HealthLake.

```
aws healthlake start-fhir-import-job \
  --input-data-config S3Uri="s3://(Bucket Name)/(Prefix Name)/" \
  --job-output-data-config '{"S3Configuration": {"S3Uri": "s3://(Bucket Name)/(Prefix Name)/", "KmsKeyId": "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-b56c-4216-a250-f4c43ef46e83"}}' \
  --datastore-id (Data store ID) \
  --data-access-role-arn "arn:aws:iam::(AWS Account ID):role/(Role Name)"
```

Output:

```
{
  "DatastoreId": "(Data store ID)",
  "JobStatus": "SUBMITTED",
  "JobId": "c145fbb27b192af392f8ce6e7838e34f"
}
```

- For API details, see [StartFHIRImportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def start_fhir_import_job(
    self,
    job_name: str,
    datastore_id: str,
    input_s3_uri: str,
    job_output_s3_uri: str,
```

```

        kms_key_id: str,
        data_access_role_arn: str,
    ) -> dict[str, str]:
        """
        Starts a HealthLake import job.
        :param job_name: The import job name.
        :param datastore_id: The data store ID.
        :param input_s3_uri: The input S3 URI.
        :param job_output_s3_uri: The job output S3 URI.
        :param kms_key_id: The KMS key ID associated with the output S3 bucket.
        :param data_access_role_arn: The data access role ARN.
        :return: The import job.
        """
        try:
            response = self.health_lake_client.start_fhir_import_job(
                JobName=job_name,
                InputDataConfig={"S3Uri": input_s3_uri},
                JobOutputDataConfig={
                    "S3Configuration": {
                        "S3Uri": job_output_s3_uri,
                        "KmsKeyId": kms_key_id,
                    }
                },
                DataAccessRoleArn=data_access_role_arn,
                DatastoreId=datastore_id,
            )
            return response
        except ClientError as err:
            logger.exception(
                "Couldn't start import job. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise

```

- For API details, see [StartFHIRImportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

1. Sign in to the [Data stores](#) page on the HealthLake Console.
2. Choose a data store.
3. Choose **Import**.

The **Import** page opens.

4. Under the **Input data** section, enter the following information:
 - **Input data location in Amazon S3**
5. Under the **Import output files** section, enter the following information:
 - **Import output files location in Amazon S3**
 - **Import output files encryption**
6. Under the **Access permissions** section, choose **Use an existing IAM service role** and select the role from the **Service role name** menu or choose **Create an IAM role**.
7. Choose **Import data**.

Note

During import, choose **Copy job ID** on the banner at the top of the page. You can use the [JobID](#) to request import job properties using the AWS CLI. For more information, see [Getting FHIR import job properties](#).

Getting FHIR import job properties

Use `DescribeFHIRImportJob` to get FHIR import job properties. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [DescribeFHIRImportJob](#) in the *AWS HealthLake API Reference*.

To get FHIR import job properties

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To describe a FHIR import job

The following `describe-fhir-import-job` example shows how to learn the properties of a FHIR import job using AWS HealthLake.

```
aws healthlake describe-fhir-import-job \
  --datastore-id (Data store ID) \
  --job-id c145fbb27b192af392f8ce6e7838e34f
```

Output:

```
{
  "ImportJobProperties": {
    "InputDataConfig": {
      "S3Uri": "s3://(Bucket Name)/(Prefix Name)/"
      { "arrayitem2": 2 }
    },
    "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role Name)",
    "JobStatus": "COMPLETED",
    "JobId": "c145fbb27b192af392f8ce6e7838e34f",
    "SubmitTime": 1606272542.161,
    "EndTime": 1606272609.497,
    "DatastoreId": "(Data store ID)"
  }
}
```

- For API details, see [DescribeFHIRImportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def describe_fhir_import_job(
    self, datastore_id: str, job_id: str
) -> dict[str, any]:
    """
    Describes a HealthLake import job.
    :param datastore_id: The data store ID.
    :param job_id: The import job ID.
    :return: The import job description.
    """
    try:
        response = self.health_lake_client.describe_fhir_import_job(
            DatastoreId=datastore_id, JobId=job_id
        )
        return response["ImportJobProperties"]
    except ClientError as err:
        logger.exception(
            "Couldn't describe import job with ID %s. Here's why %s",
            job_id,
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [DescribeFHIRImportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

 Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

 Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

 Note

FHIR import job information is not available on the HealthLake Console. Instead, use the AWS CLI with `DescribeFHIRImportJob` to request import job properties such as [JobStatus](#). For more information, refer to the AWS CLI example on this page.

Listing FHIR import jobs

Use `ListFHIRImportJobs` to list FHIR import jobs for an active HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [ListFHIRImportJobs](#) in the *AWS HealthLake API Reference*.

To list FHIR import jobs

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To list all FHIR import jobs

The following `list-fhir-import-jobs` example shows how to use the command to view a list of all import jobs associated with an account.

```
aws healthlake list-fhir-import-jobs \
  --datastore-id (Data store ID) \
  --submitted-before (DATE like 2024-10-13T19:00:00Z) \
  --submitted-after (DATE like 2020-10-13T19:00:00Z ) \
  --job-name "FHIR-IMPORT" \
  --job-status SUBMITTED \
  --max-results (Integer between 1 and 500)
```

Output:

```
{
  "ImportJobPropertiesList": [
    {
      "JobId": "c0fddb76f238297632d4aebdbfc9ddf",
      "JobStatus": "COMPLETED",
      "SubmitTime": "2024-11-20T10:08:46.813000-05:00",
      "EndTime": "2024-11-20T10:10:09.093000-05:00",
      "DatastoreId": "(Data store ID)",
      "InputDataConfig": {
        "S3Uri": "s3://(Bucket Name)/(Prefix Name)/"
      },
      "JobOutputDataConfig": {
        "S3Configuration": {
          "S3Uri": "s3://(Bucket Name)/
import/6407b9ae4c2def3cb6f1a46a0c599ec0-FHIR_IMPORT-
c0fddb76f238297632d4aebdbfc9ddf/",
          "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/b7f645cb-
e564-4981-8672-9e012d1ff1a0"
        }
      },
      "JobProgressReport": {
        "TotalNumberOfScannedFiles": 1,
```

```

        "TotalSizeOfScannedFilesInMB": 0.001798,
        "TotalNumberOfImportedFiles": 1,
        "TotalNumberOfResourcesScanned": 1,
        "TotalNumberOfResourcesImported": 1,
        "TotalNumberOfResourcesWithCustomerError": 0,
        "TotalNumberOfFilesReadWithCustomerError": 0,
        "Throughput": 0.0
    },
    "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role Name)"
}
]
}

```

- For API details, see [ListFHIRImportJobs](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def list_fhir_import_jobs(
    self,
    datastore_id: str,
    job_name: str = None,
    job_status: str = None,
    submitted_before: datetime = None,
    submitted_after: datetime = None,
) -> list[dict[str, any]]:
    """
    Lists HealthLake import jobs satisfying the conditions.

```

```

        :param datastore_id: The data store ID.
        :param job_name: The import job name.
        :param job_status: The import job status.
        :param submitted_before: The import job submitted before the specified
date.
        :param submitted_after: The import job submitted after the specified
date.
        :return: A list of import jobs.
        """
    try:
        parameters = {"DatastoreId": datastore_id}
        if job_name is not None:
            parameters["JobName"] = job_name
        if job_status is not None:
            parameters["JobStatus"] = job_status
        if submitted_before is not None:
            parameters["SubmittedBefore"] = submitted_before
        if submitted_after is not None:
            parameters["SubmittedAfter"] = submitted_after
        next_token = None
        jobs = []
        # Loop through paginated results.
        while True:
            if next_token is not None:
                parameters["NextToken"] = next_token
            response =
self.health_lake_client.list_fhir_import_jobs(**parameters)
            jobs.extend(response["ImportJobPropertiesList"])
            if "NextToken" in response:
                next_token = response["NextToken"]
            else:
                break
        return jobs
    except ClientError as err:
        logger.exception(
            "Couldn't list import jobs. Here's why %s",
            err.response["Error"]["Message"],
        )
        raise

```

- For API details, see [ListFHIRImportJobs](#) in *AWS SDK for Python (Boto3) API Reference*.

 Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

 Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

 Note

FHIR import job information is not available on the HealthLake Console. Instead, use the AWS CLI with `ListFHIRImportJobs` to list all FHIR import jobs. For more information, refer to the AWS CLI example on this page.

Managing FHIR resources in AWS HealthLake

Use FHIR R4 RESTful API interactions to manage FHIR resources in a HealthLake data store. The following sections describe all HealthLake-supported FHIR R4 RESTful API interactions available for FHIR resource management. For information about HealthLake data store capabilities and which portions of the FHIR specification it supports, see [FHIR R4 Capability Statement for AWS HealthLake](#).

Note

The FHIR interactions listed in this chapter are built in conformance to the HL7 FHIR R4 standard for health care data exchange. Because they are representations of HL7 FHIR services, they are not offered through AWS CLI and AWS SDKs. For more information, see [RESTful API](#) in the FHIR R4 RESTful API documentation.

The following table lists FHIR R4 interactions supported by AWS HealthLake. For information about FHIR *resource types* supported by HealthLake, see [Resource types](#).

FHIR R4 interactions supported by AWS HealthLake

Interaction	Description
Whole system interactions	
capabilities	Get a capability statement for the system. See FHIR R4 Capability Statement for AWS HealthLake .
batch	Update, create, or delete a set of resources in a single interaction. See Bundling FHIR resources .
Type level interactions	
create	Create a new resource with a server-assigned ID. See Creating a FHIR resource .
search	Search a resource type based on some filter criteria. See Searching FHIR resources .

Interaction	Description
<u>history</u>	Retrieve the change history for a particular resource type. See <u>Reading FHIR resource history</u> .
Instance level interactions	
<u>read</u>	Read the current state of a resource. See <u>Reading a FHIR resource</u> .
<u>history</u>	Read the change history for a particular resource. See <u>Reading FHIR resource history</u> .
<u>vread</u>	Read the state of a specific version of the resource. See <u>Reading version-specific FHIR resource history</u> .
<u>update</u>	Update a resource by its ID (or create it if it's new). See <u>Updating a FHIR resource</u> .
<u>delete</u>	Delete a resource. See <u>Deleting a FHIR resource</u> .

Topics

- [Creating a FHIR resource](#)
- [Reading a FHIR resource](#)
- [Reading FHIR resource history](#)
- [Updating a FHIR resource](#)
- [Modifying Resources with PATCH operation](#)
- [Bundling FHIR resources](#)
- [Deleting a FHIR resource](#)
- [Idempotency and Concurrency](#)

Creating a FHIR resource

The FHIR `create` interaction creates a new FHIR resource in a HealthLake data store. For additional information, see [create](#) in the FHIR R4 RESTful API documentation.

To create a FHIR resource

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Determine the type of FHIR Resource to create. For more information, see [Resource types](#).
3. Construct a URL for the request using the collected values for HealthLake region and datastoreId. Also include the FHIR Resource type to create. To view the entire URL path in the following example, scroll over the **Copy** button.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource
```

4. Construct a JSON body for the request, specifying the FHIR data for the new resource. For the purpose of this procedure, we are using a FHIR Patient resource, so save the file as create-patient.json.

```
{
  "resourceType": "Patient",
  "identifier": [
    {
      "system": "urn:oid:1.2.36.146.595.217.0.1",
      "value": "12345"
    }
  ],
  "name": [
    {
      "family": "Silva",
      "given": [
        "Ana",
        "Carolina"
      ]
    }
  ],
  "gender": "female",
  "birthDate": "1992-02-10"
}
```

5. Send the request. The FHIR create interaction uses a POST request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following examples create a FHIR Patient resource in HealthLake using either curl or the HealthLake Console. To view an entire example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```
curl --request POST \
  'https://healthlake.region.amazonaws.com/datastore/datastore-id/r4/Patient' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header 'Accept: application/json' \
  --data @create-patient.json
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\"issuer\":\"https://ehr.example.com\", \"jwks_uri\": \"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\": [\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credentials\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"], \"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"], \"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\", \"permission-v2\"]}"
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

AWS Console

Note

The HealthLake Console supports only [AWS SigV4](#) authorization.

1. Sign in to the [Run query](#) page on the HealthLake Console.
2. Under the **Query settings** section, make the following selections.
 - **Data Store ID** — choose a data store ID to generate a query string.
 - **Query type** — choose Create.
 - **Resource type** — choose the FHIR [resource type](#) to create.
 - **Request body** — construct a JSON body for the request, specifying the FHIR data for the new resource.
3. Choose **Run query**.

Configuring validation level for resource creation

When creating a FHIR resource, you can optionally specify an `x-amzn-healthlake-fhir-validation-level` HTTP header to configure a validation level for the resource. AWS HealthLake currently supports the following validation levels:

- **strict**: Resources are validated according to the profile element of the resource, or the R4 specification if no profile is present. This is the default validation level for AWS HealthLake.
- **structure-only**: Resources are validated against R4, ignoring any referenced profiles.
- **minimal**: Resources are validated minimally, ignoring certain R4 rules. Resources that fail structure checks required for search/analytics will be updated to include a warning for audit.

Resources created with the minimal validation level may be ingested into a Datastore despite failing validation required for search indexing. In this case, resources will be updated to include a Healthlake specific extension to document said failures:

```
{
```

```
"url": "http://healthlake.amazonaws.com/fhir/StructureDefinition/validation-issue",
"valueString": "{\"resourceType\":\"OperationOutcome\",\"issue\":[{\"severity\":\"error\",\"code\":\"processing\",\"details\":{\"text\":\"FHIR resource in payload failed FHIR validation rules.\"}},\"diagnostics\":{\"FHIR resource in payload failed FHIR validation rules.\"}]]}"
}
```

Additionally, the following HTTP response header will be included with a value of "true":

```
x-amzn-healthlake-validation-issues : true
```

Note

The data ingested that is malformed according the R4 specification may not be searchable as expected if these errors are present.

Reading a FHIR resource

The FHIR read interaction reads the current state of a resource in a HealthLake data store. For additional information, see [read](#) in the FHIR R4 RESTful API documentation.

To read a FHIR resource

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Determine the type of FHIR Resource to read and collect the associated id value. For more information, see [Resource types](#).
3. Construct a URL for the request using the collected values for HealthLake region and datastoreId. Also include the FHIR Resource type and its associated id. To view the entire URL path in the following example, scroll over the **Copy** button.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource/id
```

4. Send the request. The FHIR read interaction uses a GET request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following curl example reads the current state of a FHIR Patient resource in HealthLake. To view the entire example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```
curl --request GET \
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id' \
  \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header 'Accept: application/json'
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\n\"issuer\": \"https://ehr.example.com\", \"jwks_uri\": \n\"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\n\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\": [\n\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\n\"client_credential\n\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/register\", \"scopes_supported\": [\n\"openid\", \"profile\", \"launch\"], \"response_types_supported\": [\n\"code\"], \"management_endpoint\": \"https://ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\n\"S256\"], \"capabilities\": [\n\"launch-ehr\", \"sso-openid-connect\", \"client-public\", \"permission-v2\"]}"
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

AWS Console

1. Sign in to the [Run query](#) page on the HealthLake Console.
2. Under the **Query settings** section, make the following selections.
 - **Data Store ID** — choose a data store ID to generate a query string.
 - **Query type** — choose Read.
 - **Resource type** — choose the FHIR [resource type](#) to read.
 - **Resource ID** — enter the FHIR resource ID.
3. Choose **Run query**.

Reading FHIR resource history

The FHIR history interaction retrieves the history of a particular FHIR resource in a HealthLake data store. Using this interaction, you can determine how the contents of a FHIR resource have changed over time. It is also useful in coordination with audit logs to see the state of a resource before and after modification. The FHIR interactions create, update, and delete result in a historical version of the resource to be saved. For additional information, see [history](#) in the FHIR R4 RESTful API documentation.

Note

You can opt out of history for specific FHIR resource types. To opt out, create a case using [AWS Support Center Console](#). To create your case, log in to your AWS account and choose **Create case**.

To read FHIR resource history

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Determine the type of FHIR Resource to read and collect the associated id value. For more information, see [Resource types](#).

- Construct a URL for the request using the collected values for HealthLake region and datastoreId. Also include the FHIR Resource type, its associated id, and optional search parameters. To view the entire URL path in the following example, scroll over the **Copy** button.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource/id/
_history{?[parameters]}
```

HealthLake supported search parameters for FHIR history interaction

Parameter	Description
<code>_count : integer</code>	The maximum number of search results on a page. The server will return the number requested or the maximum number of search results allowed by default for the data store, whichever is lower.
<code>_since : instant</code>	Only include resource versions that were created at or after the given instant in time.
<code>_at : date(Time)</code>	Only include resource versions that were current at some point during the time period specified in the date time value. For more information, see date in the <i>HL7 FHIR RESTful API documentation</i> .

- Send the request. The FHIR history interaction uses a GET request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following `curl` example uses the `_count` search parameter to return 100 historical search results per page for a FHIR Patient resource in HealthLake. To view the entire example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```
curl --request GET \
  'https://healthlake.region.amazonaws.com/datastore/datastore-id/r4/Patient/id/
_history?_count=100' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
```



```
--header "x-amz-security-token:$AWS_SESSION_TOKEN" \
--header 'Accept: application/json'
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\n\"issuer\":\n\"https://ehr.example.com\", \n\"jwks_uri\":\n\"https://ehr.example.com/.well-known/jwks.json\", \n\"authorization_endpoint\":\n\"https://ehr.example.com/auth/authorize\", \n\"token_endpoint\":\n\"https://ehr.token.com/auth/token\", \n\"token_endpoint_auth_methods_supported\":\n[\"client_secret_basic\", \"foo\"], \n\"grant_types_supported\":\n[\"client_credential\", \"foo\"], \n\"registration_endpoint\":\n\"https://ehr.example.com/auth/register\", \n\"scopes_supported\":\n[\"openid\", \"profile\", \"launch\"], \n\"response_types_supported\":\n[\"code\"], \n\"management_endpoint\":\n\"https://ehr.example.com/user/manage\", \n\"introspection_endpoint\":\n\"https://ehr.example.com/user/introspect\", \n\"revocation_endpoint\":\n\"https://ehr.example.com/user/revoke\", \n\"code_challenge_methods_supported\":\n[\"S256\"], \n\"capabilities\":\n[\"launch-ehr\", \"sso-openid-connect\", \"client-public\", \"permission-v2\"]\n}"
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

The return content of a history interaction is contained in a FHIR resource Bundle, with type set to history. It contains the specified version history, sorted with oldest versions last, and includes deleted resources. For more information, see [Resource Bundle](#) in the FHIR R4 documentation.

Reading version-specific FHIR resource history

The FHIR `vread` interaction performs a version-specific read of a resource in a HealthLake data store. Using this interaction, you can view the contents of a FHIR resource as it was at a particular time in the past.

Note

If you use FHIR history interaction *without* `vread`, HealthLake always returns the latest version of the resource's metadata.

HealthLake declares its support for versioning in [CapabilityStatement.rest.resource.versioning](#) for each supported resource. All HealthLake data stores include `Resource.meta.versionId (vid)` on all resources.

When FHIR history interaction is enabled (by default for data stores created after 10/25/2024 or by request for older data stores), the Bundle response includes the `vid` as part of the [location](#) element. In the following example, the `vid` displays as the number 1. To view the full example, see [Example Bundle/bundle-response \(JSON\)](#).

```
"response" : {  
  "status" : "201 Created",  
  "location" : "Patient/12423/_history/1",  
  ...}
```

To read version-specific FHIR resource history

1. Collect HealthLake region and `datastoreId` values. For more information, see [Getting data store properties](#).
2. Determine the FHIR Resource type to read and collect associated `id` and `vid` values. For more information, see [Resource types](#).
3. Construct a URL for the request using the values collected for HealthLake and FHIR. To view the entire URL path in the following example, scroll over the **Copy** button.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource/id/  
_history/vid
```

4. Send the request. The FHIR history interaction uses a GET request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following vread interaction returns a single instance with the content specified for the FHIR Patient resource for the version of the resource metadata specified by the vid. To view the entire URL path in the following example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```
curl --request GET \
  'https://healthlake.region.amazonaws.com/datastore/datastore-id/r4/Patient/id/_history/vid' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header 'Accept: application/json'
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\n\"issuer\": \"https://ehr.example.com\", \"jwks_uri\": \n\"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\n\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\": [\n\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\n\"client_credential\n\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/register\", \"scopes_supported\": [\n\"openid\", \"profile\", \"launch\"], \"response_types_supported\": [\n\"code\"], \"management_endpoint\": \"https://ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\n\"S256\"], \"capabilities\": [\n\"launch-ehr\", \"sso-openid-connect\", \"client-public\", \"permission-v2\"]}"
```

```
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

Updating a FHIR resource

The FHIR update interaction creates a new current version for an existing resource or creates an initial version if no resource already exists for the given `id`. For additional information, see [update](#) in the FHIR R4 RESTful API documentation.

To update a FHIR resource

1. Collect HealthLake `region` and `datastoreId` values. For more information, see [Getting data store properties](#).
2. Determine the type of FHIR Resource to update and collect the associated `id` value. For more information, see [Resource types](#).
3. Construct a URL for the request using the collected values for HealthLake `region` and `datastoreId`. Also include the FHIR Resource type and its associated `id`. To view the entire URL path in the following example, scroll over the **Copy** button.

```
PUT https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource/id
```

4. Construct a JSON body for the request, specifying the FHIR data updates to be made. For the purpose of this procedure, save the file as `update-patient.json`.

```
{
  "id": "2de04858-ba65-44c1-8af1-f2fe69a977d9",
  "resourceType": "Patient",
  "active": true,
  "name": [
    {
      "use": "official",
      "family": "Doe",
      "given": [
        "Jane"
      ]
    }
  ],
}
```

```

    {
      "use": "usual",
      "given": [
        "Jane"
      ]
    },
    "gender": "female",
    "birthDate": "1985-12-31"
  }
}

```

5. Send the request. The FHIR update interaction uses a PUT request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following `curl` example updates a Patient resource in HealthLake. To view the entire example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```

curl --request PUT \
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header 'Accept: application/json' \
  --data @update-patient.json

```

Your request will return a 200 HTTP status code if an existing resource is *updated* or a 201 HTTP status code if a new resource is created.

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```

{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\"issuer\":\"https://ehr.example.com\", \"jwks_uri\": \"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\""}"
}

```

```

\":"https://ehr.example.com/auth/authorize\","token_endpoint\":"https://
ehr.token.com/auth/token\","token_endpoint_auth_methods_supported\":
[\"client_secret_basic\","foo\"],\"grant_types_supported\":[\"client_credentia
l\","foo\"],\"registration_endpoint\":"https://ehr.example.com/auth/
register\","scopes_supported\":[\"openid\","profile\","launch\"],
\"response_types_supported\":[\"code\"],\"management_endpoint\":"https://
ehr.example.com/user/manage\","introspection_endpoint\":"https://
ehr.example.com/user/introspect\","revocation_endpoint\":"https://
ehr.example.com/user/revoke\","code_challenge_methods_supported\":[\"S256\"],
\"capabilities\":[\"launch-ehr\","sso-openid-connect\","client-public\",
\"permission-v2\"]}"
}

```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

AWS Console

1. Sign in to the [Run query](#) page on the HealthLake Console.
2. Under the **Query settings** section, make the following selections.
 - **Data Store ID** — choose a data store ID to generate a query string.
 - **Query type** — choose Update (PUT).
 - **Resource type** — choose the FHIR [resource type](#) to update or create.
 - **Request body** — construct a JSON body for the request, specifying the FHIR data to update the resource with.
3. Choose **Run query**.

Updating FHIR resources based on conditions

Conditional update allows you to update an existing resource based on some identification search criteria, rather than by logical FHIR id. When the server processes the update, it performs a search using its standard search capabilities for the resource type, with the goal of resolving a single logical id for the request.

The action the server takes depends on how many matches it finds:

- **No matches, no id provided in the request body:** The server creates the FHIR resource.

- **No matches, id provided and resource doesn't already exist with the id:** The server treats the interaction as an Update as Create interaction.
- **No matches, id provided and already exist:** The server rejects the update with a 409 Conflict error.
- **One Match, no resource id provided OR (resource id provided and it matches the found resource):** The server performs the update against the matching resource as above where, if the resource was updated, the server SHALL return a 200 OK.
- **One Match, resource id provided but does not match resource found:** The server returns a 409 Conflict error indicating the client id specification was a problem preferably with an `OperationOutcome`
- **Multiple matches:** The server returns a 412 Precondition Failed error indicating the client's criteria were not selective enough preferably with an `OperationOutcome`

The following example updates a Patient resource whose name is peter, birthdate is 1st Jan 2000, and phone number is 1234567890.

```
PUT https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?
name=peter&birthdate=2000-01-01&phone=1234567890
```

Configuring validation level for resource updates

When updating a FHIR resource, you can optionally specify an `x-amzn-healthlake-fhir-validation-level` HTTP header to configure a validation level for the resource. AWS HealthLake currently supports the following validation levels:

- **strict:** Resources are validated according to the profile element of the resource, or the R4 specification if no profile is present. This is the default validation level for AWS HealthLake.
- **structure-only:** Resources are validated against R4, ignoring any referenced profiles.
- **minimal:** Resources are validated minimally, ignoring certain R4 rules. Resources that fail structure checks required for search/analytics will be updated to include a warning for audit.

Resources updated with the minimal validation level may be ingested into a Datastore despite failing validation required for search indexing. In this case, resources will be updated to include a Healthlake specific extension to document said failures:

```
{
```

```
"url": "http://healthlake.amazonaws.com/fhir/StructureDefinition/validation-issue",
"valueString": "{\"resourceType\":\"OperationOutcome\",\"issue\":[{\"severity\":\"error\",\"code\":\"processing\",\"details\":{\"text\":\"FHIR resource in payload failed FHIR validation rules.\"}},\"diagnostics\":{\"FHIR resource in payload failed FHIR validation rules.\"}]}"
}
```

Additionally, the following HTTP response header will be included with a value of "true":

```
x-amzn-healthlake-validation-issues : true
```

Note

Note that data ingested that is malformed according the R4 specification may not be searchable as expected if these errors are present.

Modifying Resources with PATCH operation

AWS HealthLake now supports the PATCH operation for FHIR resources, enabling you to modify resources by targeting specific elements to add, replace, or delete without updating the entire resource. This operation is particularly useful for remote clients or those with limited bandwidth looking to reduce network usage. This operation is particularly useful when you need to:

- Make targeted updates to large resources
- Reduce network bandwidth usage
- Perform atomic modifications on specific resource elements
- Minimize the risk of overwriting concurrent changes

Usage

The PATCH operation can be invoked on FHIR resources using the PATCH HTTP method:

Supported Operations

```
PATCH [base]/[resource-type]/[id]?_format=[mime-type]
```


Supported PATCH Operations

HealthLake supports the following [JSON Patch](#) operations according to RFC 6902:

Operation	Description
add	Add a new value to the resource
remove	Remove a value from the resource
replace	Replace an existing value in the resource
move	Functionally identical to a "remove" operation on the "from" location, followed immediately by an "add" operation at the target location with the value that was just removed
copy	Copy the value at a specified location to the target location
test	Test that a value at the target location is equal to a specified value

Request Headers

Header	Required	Description
Content-Type	Yes	Must be application/json-patch+json
If-Match	No	Version-specific conditional update using ETag

Examples

PATCH Request with Multiple Operations

```
PATCH [base]/Patient/example
Content-Type: application/json-patch+json
If-Match: W/"1"
```

```
[
  {
    "op": "replace",
    "path": "/name/0/family",
    "value": "Smith"
  },
  {
    "op": "add",
    "path": "/telecom/-",
    "value": {
      "system": "phone",
      "value": "*****",
      "use": "home"
    }
  },
  {
    "op": "remove",
    "path": "/address/0"
  },
  {
    "op": "move",
    "from": "/name/0/family",
    "path": "/name/1/family"
  },
  {
    "op": "test",
    "path": "/gender",
    "value": "male"
  },
  {
    "op": "copy",
    "from": "/name/0",
    "path": "/name/1"
  }
]
```

PATCH Request with Single Operation

```
PATCH [base]/Patient/example
Content-Type: application/json-patch+json
```

```
[
```

```
{
  "op": "replace",
  "path": "/active",
  "value": false
}
]
```

Sample Response

The operation returns the updated resource with new version information:

```
HTTP/1.1 200 OK
Content-Type: application/fhir+json
ETag: W/"2"
Last-Modified: Mon, 05 May 2025 10:10:10 GMT
```

```
{
  "resourceType": "Patient",
  "id": "example",
  "active": true,
  "name": [
    {
      "family": "Smith",
      "given": ["John"]
    }
  ],
  "telecom": [
    {
      "system": "phone",
      "value": "*****",
      "use": "home"
    }
  ],
  "meta": {
    "versionId": "2",
    "lastUpdated": "2025-05-05T10:10:10Z"
  }
}
```

JSON Patch Path Syntax

The path parameter uses JSON Pointer syntax (RFC 6901):

Path Example	Description
/name/0/family	First name's family element
/telecom/-	Append to telecom array
/active	Root-level active element
/address/0/ line/1	Second line of first address

Behavior

The PATCH operation:

1. Validates the JSON Patch syntax according to RFC 6902 and RFC 6901
2. Applies operations atomically - all operations succeed or all fail
3. Updates the resource version ID and creates a new history entry
4. Preserves the original resource in history before applying changes
5. Validates FHIR resource constraints after applying patches
6. Supports conditional updates using If-Match header with ETag

Error Handling

The operation handles the following error conditions:

- 400 Bad Request: Invalid patch syntax (non-conformant request or malformed JSON Patch)
- 404 Not Found: Resource not found (specified ID does not exist)
- 409 Conflict: Version conflict (concurrent updates or non-current version ID provided)
- 422 Unprocessable Entity: Patch operations cannot be applied to the specified resource elements

Caveats

- Only `application/json-patch+json` content type is supported
- Conditional PATCH operations using search conditions are not supported

For more information about PATCH operations, see the [FHIR R4 PATCH](#) documentation.

Bundling FHIR resources

A FHIR Bundle is a container for a collection of FHIR resources in AWS HealthLake. AWS HealthLake supports two types of bundles with different behaviors: [batch](#) or [transaction](#).

- For a 'batch' bundle, each FHIR resource contained in the bundle is processed and logged individually. Each resource operation is treated independently from the other resources.
- For a 'transaction' bundle, all FHIR resources contained in the bundle are processed as an atomic operation. All of the resources in the operation must succeed, or no resource updates in the bundle are committed and stored.

You can bundle FHIR resources of the same or different types, and they can include a mix of other FHIR interactions defined in this chapter (e.g. create, read, update, delete, and search). For additional information, see [Resource Bundle](#) in the FHIR R4 documentation.

Key differences between Batch and Transaction type Bundles:

Batch

- Independent operations that can succeed or fail individually
- Processing continues even if some operations fail
- Order of execution not guaranteed
- Ideal for bulk operations where partial success is acceptable

Transaction

- Atomicity guaranteed - either all succeed or all fail
- Maintains referential integrity for locally referenced (within Bundle) resources
- Operations processed in the order specified
- Fails completely if any operation fails

Example Use Cases:

- Batch: Uploading multiple unrelated patient records
- Transaction: Creating a patient with related observations and conditions where all must succeed together

Note

Both use Bundle resource type but differ in 'type' field:

```
{
  "resourceType": "Bundle",
  "type": "transaction",
  "entry": [
    {
      "fullUrl": "urn:uuid:4f6a30fb-cd3c-4ab6-8757-532101f72065",
      "resource": {
        "resourceType": "Patient",
        "id": "new-patient",
        "active": true,
        "name": [
          {
            "family": "Johnson",
            "given": [
              "Sarah"
            ]
          }
        ],
        "gender": "female",
        "birthDate": "1985-08-12",
        "telecom": [
          {
            "system": "phone",
            "value": "555-123-4567",
            "use": "home"
          }
        ]
      },
      "request": {
        "method": "POST",
        "url": "Patient"
      }
    },
    {
      "fullUrl": "urn:uuid:7f83f473-d8cc-4a8d-86d3-9d9876a3248b",
      "resource": {
        "resourceType": "Observation",
```

```
"id": "blood-pressure",
"status": "final",
"code": {
  "coding": [
    {
      "system": "http://loinc.org",
      "code": "85354-9",
      "display": "Blood pressure panel"
    }
  ],
  "text": "Blood pressure panel"
},
"subject": {
  "reference": "urn:uuid:4f6a30fb-cd3c-4ab6-8757-532101f72065"
},
"effectiveDateTime": "2023-10-15T09:30:00Z",
"component": [
  {
    "code": {
      "coding": [
        {
          "system": "http://loinc.org",
          "code": "8480-6",
          "display": "Systolic blood pressure"
        }
      ]
    },
    "valueQuantity": {
      "value": 120,
      "unit": "mmHg",
      "system": "http://unitsofmeasure.org",
      "code": "mm[Hg]"
    }
  },
  {
    "code": {
      "coding": [
        {
          "system": "http://loinc.org",
          "code": "8462-4",
          "display": "Diastolic blood pressure"
        }
      ]
    }
  }
],
}
```

```
        "valueQuantity": {
          "value": 80,
          "unit": "mmHg",
          "system": "http://unitsofmeasure.org",
          "code": "mm[Hg]"
        }
      }
    ],
    "request": {
      "method": "POST",
      "url": "Observation"
    }
  },
  {
    "resource": {
      "resourceType": "Appointment",
      "id": "appointment-123",
      "status": "booked",
      "description": "Annual physical examination",
      "start": "2023-11-15T09:00:00Z",
      "end": "2023-11-15T09:30:00Z",
      "participant": [
        {
          "actor": {
            "reference": "urn:uuid:4f6a30fb-cd3c-4ab6-8757-532101f72065"
          },
          "status": "accepted"
        }
      ]
    },
    "request": {
      "method": "PUT",
      "url": "Appointment/appointment-123"
    }
  },
  {
    "request": {
      "method": "DELETE",
      "url": "MedicationRequest/med-request-456"
    }
  }
]
```



```
}
```

Bundling FHIR resources as independent entities

To bundle FHIR resources as independent entities

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Construct a URL for the request using the collected values for HealthLake region and datastoreId. Do *not* specify a FHIR resource type in the URL. To view the entire URL path in the following example, scroll over the **Copy** button.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/
```

3. Construct a JSON body for the request, specifying each HTTP verb as part of the method elements. The following example uses a batch type interaction with the Bundle resource to create new Patient and Medication resources. All required sections are commented accordingly. For the purpose of this procedure, save the file as `batch-independent.json`.

```
{
  "resourceType": "Bundle",
  "id": "bundle-batch",
  "meta": {
    "lastUpdated": "2014-08-18T01:43:30Z"
  },
  "type": "batch",
  "entry": [
    {
      "resource": {
        "resourceType": "Patient",
        "meta": {
          "lastUpdated": "2022-06-03T17:53:36.724Z"
        },
        "text": {
          "status": "generated",
          "div": "Some narrative"
        },
        "active": true,
        "name": [
          {
```

```

        "use": "official",
        "family": "Jackson",
        "given": [
            "Mateo",
            "James"
        ]
    },
    ],
    "gender": "male",
    "birthDate": "1974-12-25"
},
"request": {
    "method": "POST",
    "url": "Patient"
}
},
{
    "resource": {
        "resourceType": "Medication",
        "id": "med0310",
        "contained": [
            {
                "resourceType": "Substance",
                "id": "sub03",
                "code": {
                    "coding": [
                        {
                            "system": "http://snomed.info/sct",
                            "code": "55452001",
                            "display": "Oxycodone (substance)"
                        }
                    ]
                }
            }
        ],
        "code": {
            "coding": [
                {
                    "system": "http://snomed.info/sct",
                    "code": "430127000",
                    "display": "Oral Form Oxycodone (product)"
                }
            ]
        }
    },

```

```

        "form": {
            "coding": [
                {
                    "system": "http://snomed.info/sct",
                    "code": "385055001",
                    "display": "Tablet dose form (qualifier value)"
                }
            ]
        },
        "ingredient": [
            {
                "itemReference": {
                    "reference": "#sub03"
                },
                "strength": {
                    "numerator": {
                        "value": 5,
                        "system": "http://unitsofmeasure.org",
                        "code": "mg"
                    },
                    "denominator": {
                        "value": 1,
                        "system": "http://terminology.hl7.org/CodeSystem/
v3-orderableDrugForm",
                        "code": "TAB"
                    }
                }
            }
        ],
        "request": {
            "method": "POST",
            "url": "Medication"
        }
    }
}

```

4. Send the request. The FHIR Bundle batch type uses a POST request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following code example uses the `curl` command line tool for demonstration purposes.

SigV4

SigV4 authorization

```
curl --request POST \
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header 'Accept: application/json' \
  --data @batch-type.json
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\"issuer\":\"https://ehr.example.com\", \"jwks_uri\": \"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\": [\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credentials\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"], \"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"], \"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\", \"permission-v2\"]}"
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

The server returns a response showing the Patient and Medication resources created as a result of the Bundle batch type request.

Conditional PUTs in Bundles

AWS HealthLake supports conditional updates within Bundles using the following query parameters:

- `_id` (standalone)
- `_id` in combination with one of the following:
 - `_tag`
 - `_createdAt`
 - `_lastUpdated`

Based on the results of matching the conditions provided to the existing resource, the following will occur with the associated result codes indicating the action taken:

When creating or updating FHIR resources, the system handles different scenarios based on resource ID provision and existing matches:

- Resources without IDs are always created (201).
- Resources with new IDs are created (201).
- Resources with existing IDs either update the matching resource (200) or return errors if there's a conflict (409) or ID mismatch (400).
- Multiple matching resources trigger a precondition failure (419).

In the example Bundle with conditional update, the Patient resource with FHIR ID 456 will only update if the condition `_lastUpdated=1t2025-04-20` is met.

```
{
  "resourceType": "Bundle",
  "id": "bundle-batch",
  "meta": {
    "lastUpdated": "2014-08-18T01:43:30Z"
  },
}
```

```
"type": "batch",
"entry": [
  {
    "resource": {
      "resourceType": "Patient",
      "id": "476",
      "meta": {
        "lastUpdated": "2022-06-03T17:53:36.724Z"
      },
      "active": true,
      "name": [
        {
          "use": "official",
          "family": "Jackson",
          "given": [
            "Mateo",
            "James"
          ]
        }
      ],
      "gender": "male",
      "birthDate": "1974-12-25"
    },
    "request": {
      "method": "PUT",
      "url": "Patient?_id=476&_lastUpdated=lt2025-04-20"
    }
  },
  {
    "resource": {
      "resourceType": "Medication",
      "id": "med0310",
      "contained": [
        {
          "resourceType": "Substance",
          "id": "sub03",
          "code": {
            "coding": [
              {
                "system": "http://snomed.info/sct",
                "code": "55452001",
                "display": "Oxycodone (substance)"
              }
            ]
          }
        }
      ]
    }
  }
]
```

```

        }
    },
    ],
    "code": {
        "coding": [
            {
                "system": "http://snomed.info/sct",
                "code": "430127000",
                "display": "Oral Form Oxycodone (product)"
            }
        ]
    },
    },
    "form": {
        "coding": [
            {
                "system": "http://snomed.info/sct",
                "code": "385055001",
                "display": "Tablet dose form (qualifier value)"
            }
        ]
    },
    },
    "ingredient": [
        {
            "itemReference": {
                "reference": "#sub03"
            },
            "strength": {
                "numerator": {
                    "value": 5,
                    "system": "http://unitsofmeasure.org",
                    "code": "mg"
                },
                "denominator": {
                    "value": 1,
                    "system": "http://terminology.hl7.org/CodeSystem/v3-
orderableDrugForm",
                    "code": "TAB"
                }
            }
        }
    ],
    },
    "request": {
        "method": "POST",

```

```

        "url": "Medication"
      }
    }
  ]
}

```

Bundling FHIR resources as a single entity

To bundle FHIR resources as a single entity

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Construct a URL for the request using the collected values for HealthLake region and datastoreId. Include the FHIR resource type Bundle as part of the URL. To view the entire URL path in the following example, scroll over the **Copy** button.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Bundle
```

3. Construct a JSON body for the request, specifying the FHIR resources to group together. The following example groups two Patient resources in HealthLake. For the purpose of this procedure, save the file as batch-single.json.

```

{
  "resourceType": "Bundle",
  "id": "bundle-minimal",
  "language": "en-US",
  "identifier": {
    "system": "urn:oid:1.2.3.4.5",
    "value": "28b95815-76ce-457b-b7ae-a972e527db4f"
  },
  "type": "document",
  "timestamp": "2020-12-11T14:30:00+01:00",
  "entry": [
    {
      "fullUrl": "urn:uuid:f40b07e3-37e8-48c3-bf1c-ae70fe12dabf",
      "resource": {
        "resourceType": "Composition",
        "id": "f40b07e3-37e8-48c3-bf1c-ae70fe12dabf",
        "status": "final",
        "type": {

```



```

        "coding": [
            {
                "system": "http://loinc.org",
                "code": "60591-5",
                "display": "Patient summary Document"
            }
        ],
        "date": "2020-12-11T14:30:00+01:00",
        "author": [
            {
                "reference":
"urn:uuid:45271f7f-63ab-4946-970f-3daaaa0663ff"
            }
        ],
        "title": "Patient Summary as of December 7, 2020 14:30"
    },
    {
        "fullUrl": "urn:uuid:45271f7f-63ab-4946-970f-3daaaa0663ff",
        "resource": {
            "resourceType": "Practitioner",
            "id": "45271f7f-63ab-4946-970f-3daaaa0663ff",

            "active": true,
            "name": [
                {
                    "family": "Doe",
                    "given": [
                        "John"
                    ]
                }
            ]
        }
    }
]
}

```

4. Send the request. The FHIR Bundle document type uses a POST request with [AWS Signature Version 4](#) signing protocol. The following code example uses the `curl` command line tool for demonstration purposes.

SigV4

SigV4 authorization

```
curl --request POST \
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Bundle' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header 'Accept: application/json' \
  --data @document-type.json
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\"issuer\":\"https://ehr.example.com\", \"jwks_uri\": \"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\": [\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credentials\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"], \"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"], \"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\", \"permission-v2\"]}"
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

The server returns a response showing two Patient resources created as a result of the Bundle document type request.

Configuring validation level for bundles

When bundling FHIR resources, you can optionally specify an `x-amzn-healthlake-fhir-validation-level` HTTP header to configure a validation level for the resource. This validation level will be set for all create and update requests within the bundle. AWS HealthLake currently supports the following validation levels:

- **strict**: Resources are validated according to the profile element of the resource, or the R4 specification if no profile is present. This is the default validation level for AWS HealthLake.
- **structure-only**: Resources are validated against R4, ignoring any referenced profiles.
- **minimal**: Resources are validated minimally, ignoring certain R4 rules. Resources that fail structure checks required for search/analytics will be updated to include a warning for audit.

Resources bundled with the minimal validation level may be ingested into a Datastore despite failing validation required for search indexing. In this case, resources will be updated to include a Healthlake specific extension to document said failures, and the entries within the Bundle response will include `OperationOutcome` resources as follows:

```
{
  "resourceType": "Bundle",
  "type": "batch-response",
  "timestamp": "2025-08-25T22:58:48.846287342Z",
  "entry": [
    {
      "response": {
        "status": "201",
        "location": "Patient/195abc49-ba8e-4c8b-95c2-abc88fef7544/_history/1",
        "etag": "W/\"1\"",
        "lastModified": "2025-08-25T22:58:48.801245445Z",
        "outcome": {
          "resourceType": "OperationOutcome",
          "issue": [
            {
              "severity": "error",
```



```
DELETE https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource/id
```

4. Send the request. The FHIR delete interaction uses a DELETE request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following curl example removes an existing FHIR Patient resource from a HealthLake data store. To view the entire example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```
curl --request DELETE \
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header 'Accept: application/json'
```

The server returns a 204 HTTP status code confirming the resource has been removed from the HealthLake data store. If a delete request fails, you will receive a 400 series HTTP status code indicating why the request failed.

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\n\"issuer\": \"https://ehr.example.com\", \"jwks_uri\": \"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\": [\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credential\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"], \"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://
```

```
ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://  
ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://  
ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"],  
\"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\",  
\"permission-v2\"]}]\"  
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

AWS Console

1. Sign in to the [Run query](#) page on the HealthLake Console.
2. Under the **Query settings** section, make the following selections.
 - **Data Store ID** — choose a data store ID to generate a query string.
 - **Query type** — choose Delete.
 - **Resource type** — choose the FHIR [resource type](#) to delete.
 - **Resource ID** — enter the FHIR resource ID.
3. Choose **Run query**.

Deleting FHIR resources based on conditions

Conditional delete is particularly useful when you don't know the specific FHIR resource ID but have other identifying information about the resource you want to delete.

Conditional delete allows you to delete an existing resource based on search criteria rather than by logical FHIR ID. When the server processes the delete request, it performs a search using standard search capabilities for the resource type to resolve a single logical ID for the request.

How conditional delete works

The server's action depends on how many matches it finds:

1. **No matches:** The server attempts an ordinary delete and responds appropriately (404 Not Found for non-existent resource, 204 No Content for already deleted resource)
2. **One match:** The server performs an ordinary delete on the matching resource

3. **Multiple matches:** Returns a 412 Precondition Failed error indicating the client's criteria were not selective enough

Response scenarios

AWS HealthLake handles conditional delete operations with the following response patterns:

Successful Operations

- When your search criteria successfully identify a single active resource, the system returns **204 No Content** after completing the deletion, just like standard delete operations.

ID-Based Conditional Delete

When performing conditional delete based on `id` with additional parameters (`createdAt`, `tag`, or `_lastUpdated`):

- **204 No Content:** Resource was already deleted
- **404 Not Found:** Resource doesn't exist
- **409 Conflict:** ID matches but other parameters don't match

Non-ID-Based Conditional Delete

When `id` is not provided or when using parameters other than `createdAt`, `tag`, or `_lastUpdated`:

- **404 Not Found:** No matches found

Conflict Situations

Several scenarios result in 412 Precondition Failed responses:

- Multiple resources match your search criteria (criteria not specific enough)
- Version conflicts when using ETag headers with `If-Match`
- Resource updates occurring between search and delete operations

Example of a Successful Conditional Delete

The following example deletes a Patient resource based on specific criteria:

```
DELETE https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?
name=peter&birthdate=2000-01-01&phone=1234567890
```

This request deletes a Patient resource where:

- Name is "peter"
- Birth date is January 1, 2000
- Phone number is 1234567890

Best Practices

1. Use specific search criteria to avoid multiple matches and prevent 412 errors.
2. Consider ETag headers for version control when needed to handle concurrent modifications.
3. Handle error responses appropriately:
 - For 404: Refine your search criteria
 - For 412: Make criteria more specific or resolve version conflicts
4. Prepare for timing conflicts in high-concurrency environments where resources may be modified between search and delete operations.

Idempotency and Concurrency

Idempotency Keys

AWS HealthLake supports idempotency keys for FHIR POST operations, providing a robust mechanism to ensure data integrity during resource creation. By including a unique UUID as an idempotency key in the request header, healthcare applications can guarantee that each FHIR resource is created exactly once, even in scenarios involving network instability or automatic retries.

This feature is particularly crucial for healthcare systems where duplicate medical records could have serious consequences. When a request is received with the same idempotency key as a

previous request, HealthLake will return the original resource instead of creating a duplicate. For example, this could occur during a retry loop or due to redundant request pipelines. Using the idempotency key allows HealthLake to maintain data consistency while providing a seamless experience for client applications handling intermittent connectivity issues.

Implementation

```
POST /<baseURL>/Patient
x-amz-fhir-idempotency-key: 123e4567-e89b-12d3-a456-426614174000
{
  "resourceType": "Patient",
  "name": [...]
}
```

Response Scenarios

First Request (201 Created)

- New resource created successfully
- Response includes resource ID

Duplicate Request (409 Conflict)

- Same idempotency key detected
- Original resource returned
- No new resource created

Invalid Request (400 Bad Request)

- Malformed UUID
- Missing required fields

Best Practices

- Generate unique UUID for each new resource creation
- Store idempotency keys for retry logic
- Use consistent key format: UUID v4 recommended
- Implement in client applications handling resource creation

Note

This feature is particularly valuable for healthcare systems requiring strict data accuracy and preventing duplicate medical records.

ETag in AWS HealthLake

AWS HealthLake uses ETags for optimistic concurrency control in FHIR resources, providing a reliable mechanism to manage concurrent modifications and maintain data consistency. An ETag is a unique identifier that represents a specific version of a resource, functioning as a version control system through HTTP headers. When reading or modifying resources, applications can use ETags to prevent unintended overwrites and ensure data integrity, particularly in scenarios with potential concurrent updates.

Implementation Example

```
// Initial Read
GET /fhir/Patient/123
Response:
ETag: W/"1"

// Update with If-Match
PUT /fhir/Patient/123
If-Match: W/"1"
{resource content}

// Create with If-None-Match
PUT /fhir/Patient/123
If-None-Match: *
{resource content}
// Succeeds only if resource doesn't exist
// Fails with 412 if resource exists
```

Response Scenarios

Successful Operation (200 OK or 204 No Content)

- ETag matches current version
- Operation proceeds as intended

Version Conflict (412 Precondition Failed)

- ETag doesn't match current version
- Update rejected to prevent data loss

Best Practices

- Include ETags in all update and delete operations
- Implement retry logic for handling version conflicts
- Use If-None-Match: * for create-if-not-exists scenarios
- Always verify ETag freshness before modifications

This concurrency control system is essential for maintaining the integrity of healthcare data, especially in environments with multiple users or systems accessing and modifying the same resources.

Searching FHIR resources in AWS HealthLake

Use FHIR [search](#) interaction to search a set of FHIR resources in a HealthLake data store based on some filter criteria. The `search` interaction can be performed using either a GET or POST request. For searches that involve personally identifiable information (PII) or protected health information (PHI), it's recommended to use POST requests, as PII and PHI is added as part of the request body and is encrypted in transit.

Note

The FHIR search interaction described in this chapter is built in conformance to the HL7 FHIR R4 standard for health care data exchange. Because it is a representation of a HL7 FHIR service, it is not offered through AWS CLI and AWS SDKs. For more information, see [search](#) in the FHIR R4 RESTful API documentation.

HealthLake supports a subset of FHIR R4 search parameters. For more information, see [FHIR R4 search parameters for HealthLake](#).

Topics

- [Searching FHIR resources with GET](#)
- [Searching FHIR resources with POST](#)
- [FHIR Search Consistency Levels](#)

Searching FHIR resources with GET

You can use GET requests to search a HealthLake data store. When using GET, HealthLake supports providing search parameters as part of the URL, but not as part of a request body. For more information, see [FHIR R4 search parameters for HealthLake](#).

Important

For searches that involve personally identifiable information (PII) or protected health information (PHI), security best practices call for using POST requests, as PII and PHI is added as part of the request body and is encrypted in transit. For more information, see [Searching FHIR resources with POST](#).

The following procedure is followed by examples that use GET to search a HealthLake data store.

To search a HealthLake data store with GET

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Determine the type of FHIR resource to search for and collect the associated id value. For more information, see [Resource types](#).
3. Construct a URL for the request using the collected values for HealthLake region and datastoreId. Also include the FHIR Resource type and supported [search parameters](#). To view the entire URL path in the following example, scroll over the **Copy** button.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource{?  
[parameters]}[&_format=[mime-type]]}
```

4. Send the GET request with either [AWS Signature Version 4](#) or SMART on FHIR authorization. The following curl example returns the total number of Patient resources in a HealthLake data store. To view the entire example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```
curl --request GET \  
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?  
_total=accurate' \  
  --aws-sigv4 'aws:amz:region:healthlake' \  
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \  
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \  
  --header 'Accept: application/json'
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{  
  "AuthorizationStrategy": "SMART_ON_FHIR",  
  "FineGrainedAuthorizationEnabled": true,
```

```

    "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-
lambda-name",
    "Metadata": "{ \"issuer\": \"https://ehr.example.com\", \"jwks_uri\":
\"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint
\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://
ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\":
[\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credential
\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/
register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"],
\"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://
ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://
ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://
ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"],
\"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\",
\"permission-v2\"] }"
}

```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

AWS Console

Note

The HealthLake Console supports only SigV4 authorization. SMART on FHIR authorization is supported through AWS CLI and AWS SDKs.

1. Sign in to the [Run query](#) page on the HealthLake Console.
2. Under the **Query settings** section, make the following selections.
 - **Data Store ID** — choose a data store ID to generate a query string.
 - **Query type** — choose Search with GET.
 - **Resource type** — choose the FHIR [resource type](#) to search on.
 - **Search parameters** — Select a [search parameter](#) or combination of search parameters to focus your query on specific records.
3. Choose **Run query**.

Examples: search with GET

The following tabs provide examples for searching on specific FHIR resource types with GET. The examples show how to specify search parameters in the request URLs.

Note

The HealthLake Console supports only SigV4 authorization. SMART on FHIR authorization is supported through AWS CLI and AWS SDKs.

HealthLake supports a subset of FHIR R4 search parameters. For more information, see [Search parameters](#).

Patient (age)

Although age is not a defined resource type in FHIR, it is captured as an element in the [Patient](#) resource type. Use the following example to make a GET-based search request on [Patient](#) resource types using the [birthDate](#) element and the eq [search comparator](#) to search for individuals born in the year 1997.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?
birthdate=eq1997
```

Condition

Use the following example to make a GET request on the [Condition](#) resource type. The search finds conditions in your HealthLake data store that contain the SNOMED medical code 72892002, which translates to Normal pregnancy.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Condition?
code=72892002
```

DocumentationReference

The following example shows how to create a GET request on the [DocumentReference](#) resource type for Patient(s) with a streptococcal diagnosis and who have also been prescribed amoxicillin.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/  
DocumentReference?_lastUpdated=1e2021-12-19&infer-icd10cm-entity-text-concept-  
score;=streptococcal|0.6&infer-rxnorm-entity-text-concept-score=Amoxicillin|0.8
```

Location

Use the following example to make a GET request on the [Location](#) resource type. The following search finds locations in your HealthLake data store that contain the city name Boston as part of the address.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Location?  
address=boston
```

Observation

Use the following example to make a GET-based search request on the [Observation](#) resource type. This search uses the value-concept [search parameter](#) to look for medical code 266919005, which translates to Never smoker.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Observation?  
value-concept=266919005
```

Searching FHIR resources with POST

You can use the FHIR [search](#) interaction with POST requests to search a HealthLake data store. When using POST, HealthLake supports search parameters in either the URL or in a request body, but you cannot use both in a single request.

Important

For searches that involve personally identifiable information (PII) or protected health information (PHI), security best practices call for using POST requests, as PII and PHI is added as part of the request body and is encrypted in transit.

The following procedure is followed by examples using FHIR R4 search interaction with POST to search a HealthLake data store. The examples show how to specify search parameters in the JSON request body.

To search a HealthLake data store with POST

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Determine the type of FHIR resource to search for and collect the associated id value. For more information, see [Resource types](#).
3. Construct a URL for the request using the collected values for HealthLake region and datastoreId. Also include the FHIR Resource type and `_search` interaction. To view the entire URL path in the following example, scroll over the **Copy** button.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource/  
_search
```

4. Construct a JSON body for the request, specifying the FHIR data to search for. For the purpose of this procedure, you will search Observation resources to discover patients who have never smoked. To specify the medical code status Never smoker, set `value-concept=266919005` in the JSON request body. Save the file as `search-observation.json`.

```
value-concept=266919005
```

5. Send the request. The FHIR search interaction uses the GET request with either [AWS Signature Version 4](#) or SMART on FHIR authorization.

Note

When making a POST request with search parameters in the request body, use `Content-Type: application/x-www-form-urlencoded` as part of the header.

The following `curl` example makes a POST-based search request on the Observation resource type. The request uses the [value-concept](#) search parameter to look for medical code 266919005 which indicates value Never smoker. To view the entire example, scroll over the **Copy** button.

SigV4

SigV4 authorization

```
curl --request POST \
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Observation/_search' \
  --aws-sigv4 'aws:amz:region:healthlake' \
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \
  --header "Content-Type: application/x-www-form-urlencoded" \
  --header "Accept: application/json" \
  --data @search-observation.json
```

SMART on FHIR

SMART on FHIR authorization example for the [IdentityProviderConfiguration](#) data type.

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-lambda-name",
  "Metadata": "{\n\"issuer\": \"https://ehr.example.com\", \"jwks_uri\": \"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\": [\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credentials\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"], \"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"], \"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\", \"permission-v2\"]}"
}
```

The caller can assign permissions in the authorization lambda. For more information, see [OAuth 2.0 scopes](#).

Examples: search with POST

The following tabs provide examples for searching on specific FHIR resource types with POST. The examples show how to specify a request in the URLs.

Note

The HealthLake Console supports only SigV4 authorization. SMART on FHIR authorization is supported through AWS CLI and AWS SDKs.

HealthLake supports a subset of FHIR R4 search parameters. For more information, see [Search parameters](#).

Patient (age)

Although age is not a defined resource type in FHIR, it is captured as an element in the [Patient](#) resource type. Use the following example to make a POST-based search request on the Patient resource type. The following search example uses the eq [search comparator](#) to search for individuals born in 1997.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/_search
```

To specify the year 1997 in the search, add the following element to the request body.

```
birthdate=eq1997
```

Condition

Using the following to make a POST request on the Condition resource type. This search finds locations in your HealthLake data store that contain the medical code 72892002.

You have to specify a request URL and a request body. Here is an example request URL.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Condition/_search
```

To specify the medical code you want to search, you add the following JSON element to the request body.

```
code=72892002
```

DocumentReference

To see the results of HealthLake's integrated natural language processing (NLP) when making a POST request on the DocumentReference resource type, format a request as follows.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/  
DocumentReference/_search
```

To specify the DocumentReference search parameters to reference, see [Search parameter types](#). The following query string uses multiple search parameters to search on Amazon Comprehend Medical API operations used to generate the integrated NLP results.

```
_lastUpdated=1e2021-12-19&infer-icd10cm-entity-text-concept-score;=streptococcal|  
0.6&infer-rxnorm-entity-text-concept-score=Amoxicillin|0.8
```

Location

Use the following example to make a POST request on the Location resource type. The search finds locations in your HealthLake data store that contain the city name Boston as part of the address.

You must specify a request URL and a request body. Here is an example request URL.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Location/  
_search
```

To specify Boston in the search, add the following element to the request body:

```
address=Boston
```

Observation

Use the following example to make a POST-based search request on the Observation resource type. The search uses the value-concept search parameter to look for medical code, 266919005 that indicates Never smoker.

```
POST https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Observation/  
_search
```

To specify the status, `Never smoker`, set `value-concept=266919005` in the body of the JSON.

```
value-concept=266919005
```

FHIR Search Consistency Levels

AWS HealthLake's search index operates on an Eventual Consistency model for GET and POST with SEARCH operations. With eventual consistency, if there is a pending search index update for a resource, the search results exclude version N-1 of the resource until the index update completes.

AWS HealthLake now includes the ability to select how the consistency model will behave for updated resources. Developers can include either 'Eventual Consistency', the default behavior described above or 'Strong Consistency'. Strong Consistency will allow the N-1 version of the resource for resources with pending search index updates to be included in the search results. This can be used for use case scenarios where all resources are required in the result even when the search index update has not yet completed. Customers can control this behavior using the `x-amz-fhir-history-consistency-level` request header.

Consistency levels

Strong consistency

Set `x-amz-fhir-history-consistency-level: strong` to return all matching records, including those with pending search index updates. Use this option when you need to search for resources immediately after updates.

Eventual consistency

Set `x-amz-fhir-history-consistency-level: eventual` to return only records that have completed search index updates. This is the default behavior if no consistency level is specified.

Usage example

1. When updating a resource:

```
POST <baseUrl>/Patient
```

```
Content-Type: application/fhir+json
x-amz-fhir-history-consistency-level: strong

{
  "resourceType": "Patient",
  "id": "123",
  "meta": {
    "profile": ["http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient"]
  },
  "identifier": [
    {
      "system": "http://example.org/identifiers",
      "value": "123"
    }
  ],
  "active": true,
  "name": [
    {
      "family": "Smith",
      "given": ["John"]
    }
  ],
  "gender": "male",
  "birthDate": "1970-01-01"
}
```

2. Subsequent search:

```
GET <baseUrl>/Patient?_id=123
```

Best practices

- Use strong consistency when you need to immediately search for recently updated resources
- Use eventual consistency for general queries where immediate visibility isn't critical
- Consider the trade-off between immediate visibility and potential performance impact

Note

The consistency level setting affects how quickly updated resources appear in search results but does not impact the actual storage of the resources.

Setting the optional `x-amz-fhir-history-consistency-level` header to 'strong' doubles the write capacity consumption per resource.

This feature is only applicable for data stores that have version history enabled (all datastores created after Oct 25, 2024 have it enabled by default).

Exporting FHIR data with AWS HealthLake

Use native AWS HealthLake actions to start, describe, and list FHIR export jobs. You can queue export jobs. The asynchronous export jobs are processed in a FIFO (First In First Out) manner. You can queue jobs the same way you start export jobs. If one is in progress, it will simply queue up. You can create, read, update, or delete FHIR resources while an export job is in progress.

Note

You can also export FHIR data from a HealthLake data store using the FHIR R4 `$export` operation. For more information, see [Exporting HealthLake data with FHIR \\$export](#).

Topics

- [Starting a FHIR export job](#)
- [Getting FHIR export job properties](#)
- [Listing FHIR export jobs](#)

Starting a FHIR export job

Use `StartFHIRExportJob` to start a FHIR export job from a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [StartFHIRExportJob](#) in the *AWS HealthLake API Reference*.

Note

HealthLake supports the [FHIR R4 specification](#) for health care data exchange. Therefore, all health data is exported in FHIR R4 format.

To start a FHIR export job

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To start a FHIR export job

The following `start-fhir-export-job` example shows how to start a FHIR export job using AWS HealthLake.

```
aws healthlake start-fhir-export-job \
  --output-data-config '{"S3Configuration": {"S3Uri": "s3://(Bucket Name)/\
  (Prefix Name)/", "KmsKeyId": "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-\
  b56c-4216-a250-f4c43ef46e83"}}' \
  --datastore-id (Data store ID) \
  --data-access-role-arn arn:aws:iam::(AWS Account ID):role/(Role Name)
```

Output:

```
{
  "DatastoreId": "(Data store ID)",
  "JobStatus": "SUBMITTED",
  "JobId": "9b9a51943afaedd0a8c0c26c49135a31"
}
```

- For API details, see [StartFHIRExportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
```

```

        """
        health_lake_client = boto3.client("healthlake")
        return cls(health_lake_client)

    def start_fhir_export_job(
        self,
        job_name: str,
        datastore_id: str,
        output_s3_uri: str,
        kms_key_id: str,
        data_access_role_arn: str,
    ) -> dict[str, str]:
        """
        Starts a HealthLake export job.
        :param job_name: The export job name.
        :param datastore_id: The data store ID.
        :param output_s3_uri: The output S3 URI.
        :param kms_key_id: The KMS key ID associated with the output S3 bucket.
        :param data_access_role_arn: The data access role ARN.
        :return: The export job.
        """
        try:
            response = self.health_lake_client.start_fhir_export_job(
                OutputDataConfig={
                    "S3Configuration": {"S3Uri": output_s3_uri, "KmsKeyId":
kms_key_id}
                },
                DataAccessRoleArn=data_access_role_arn,
                DatastoreId=datastore_id,
                JobName=job_name,
            )

            return response
        except ClientError as err:
            logger.exception(
                "Couldn't start export job. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise

```

- For API details, see [StartFHIRExportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

1. Sign in to the [Data stores](#) page on the HealthLake Console.
2. Choose a data store.
3. Choose **Export**.

The **Export** page opens.

4. Under the **Output data** section, enter the following information:
 - **Output data location in Amazon S3**
 - **Output encryption**
5. Under the **Access permissions** section, choose **Use an existing IAM service role** and select the role from the **Role name** menu or choose **Create an IAM role**.
6. Choose **Export data**.

Note

During export, choose **Copy job ID** on the banner at the top of the page. You can use the [JobID](#) to request export job properties using the AWS CLI. For more information, see [Getting FHIR export job properties](#).

Getting FHIR export job properties

Use `DescribeFHIRExportJob` to get export job properties from a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [DescribeFHIRExportJob](#) in the *AWS HealthLake API Reference*.

Note

HealthLake supports the [FHIR R4 specification](#) for health care data exchange. Therefore, all health data is exported in FHIR R4 format.

To describe a FHIR export job

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To describe a FHIR export job

The following `describe-fhir-export-job` example shows how to find the properties of a FHIR export job in AWS HealthLake.

```
aws healthlake describe-fhir-export-job \
  --datastore-id (Data store ID) \
  --job-id 9b9a51943afaedd0a8c0c26c49135a31
```

Output:

```
{
  "ExportJobProperties": {
    "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role Name)",
    "JobStatus": "IN_PROGRESS",
    "JobId": "9009813e9d69ba7cf79bcb3468780f16",
    "SubmitTime": "2024-11-20T11:31:46.672000-05:00",
```

```

        "EndTime": "2024-11-20T11:34:01.636000-05:00",
        "OutputDataConfig": {
            "S3Configuration": {
                "S3Uri": "s3://(Bucket Name)/(Prefix Name)/",
                "KmsKeyId": "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-
b56c-4216-a250-f4c43ef46e83"
            }
        },
        "DatastoreId": "(Data store ID)"
    }
}

```

- For API details, see [DescribeFHIRExportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def describe_fhir_export_job(
    self, datastore_id: str, job_id: str
) -> dict[str, any]:
    """
    Describes a HealthLake export job.
    :param datastore_id: The data store ID.
    :param job_id: The export job ID.
    :return: The export job description.
    """
    try:

```

```
response = self.health_lake_client.describe_fhir_export_job(
    DatastoreId=datastore_id, JobId=job_id
)
return response["ExportJobProperties"]
except ClientError as err:
    logger.exception(
        "Couldn't describe export job with ID %s. Here's why %s",
        job_id,
        err.response["Error"]["Message"],
    )
    raise
```

- For API details, see [DescribeFHIRExportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

Note

FHIR export job information is not available on the HealthLake Console. Instead, use the AWS CLI with `DescribeFHIRExportJob` to request export job properties such as [JobStatus](#). For more information, refer to the AWS CLI example on this page.

Listing FHIR export jobs

Use `ListFHIRExportJobs` to list FHIR export jobs for a HealthLake data store. The following menus provide a procedure for the AWS Management Console and code examples for the AWS CLI and AWS SDKs. For more information, see [ListFHIRExportJobs](#) in the *AWS HealthLake API Reference*.

Note

HealthLake supports the [FHIR R4 specification](#) for health care data exchange. Therefore, all health data is exported in FHIR R4 format.

To list FHIR export jobs

Choose a menu based on your access preference to AWS HealthLake.

AWS CLI and SDKs

CLI

AWS CLI

To list all FHIR export jobs

The following `list-fhir-export-jobs` example shows how to use the command to view a list of export jobs associated with an account.

```
aws healthlake list-fhir-export-jobs \
  --datastore-id (Data store ID) \
  --submitted-before (DATE like 2024-10-13T19:00:00Z) \
  --submitted-after (DATE like 2020-10-13T19:00:00Z ) \
  --job-name "FHIR-EXPORT" \
  --job-status SUBMITTED \
  --max-results (Integer between 1 and 500)
```

Output:

```
{
  "ExportJobPropertiesList": [
```

```

    {
        "ExportJobProperties": {
            "OutputDataConfig": {
                "S3Uri": "s3://(Bucket Name)/(Prefix Name)/",
                "S3Configuration": {
                    "S3Uri": "s3://(Bucket Name)/(Prefix Name)/",
                    "KmsKeyId": "(KmsKey Id)"
                }
            },
            "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role
Name)",
            "JobStatus": "COMPLETED",
            "JobId": "c145fbb27b192af392f8ce6e7838e34f",
            "JobName": "FHIR-EXPORT",
            "SubmitTime": "2024-11-20T11:31:46.672000-05:00",
            "EndTime": "2024-11-20T11:34:01.636000-05:00",
            "DatastoreId": "(Data store ID)"
        }
    }
]
}

```

- For API details, see [ListFHIRExportJobs](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def list_fhir_export_jobs(

```



```

        self,
        datastore_id: str,
        job_name: str = None,
        job_status: str = None,
        submitted_before: datetime = None,
        submitted_after: datetime = None,
    ) -> list[dict[str, any]]:
        """
        Lists HealthLake export jobs satisfying the conditions.
        :param datastore_id: The data store ID.
        :param job_name: The export job name.
        :param job_status: The export job status.
        :param submitted_before: The export job submitted before the specified
date.
        :param submitted_after: The export job submitted after the specified
date.
        :return: A list of export jobs.
        """
        try:
            parameters = {"DatastoreId": datastore_id}
            if job_name is not None:
                parameters["JobName"] = job_name
            if job_status is not None:
                parameters["JobStatus"] = job_status
            if submitted_before is not None:
                parameters["SubmittedBefore"] = submitted_before
            if submitted_after is not None:
                parameters["SubmittedAfter"] = submitted_after
            next_token = None
            jobs = []
            # Loop through paginated results.
            while True:
                if next_token is not None:
                    parameters["NextToken"] = next_token
                response =
self.health_lake_client.list_fhir_export_jobs(**parameters)
                jobs.extend(response["ExportJobPropertiesList"])
                if "NextToken" in response:
                    next_token = response["NextToken"]
                else:
                    break
            return jobs
        except ClientError as err:
            logger.exception(

```

```
        "Couldn't list export jobs. Here's why %s",  
        err.response["Error"]["Message"],  
    )  
    raise
```

- For API details, see [ListFHIRExportJobs](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Example availability

Can't find what you need? Request a code example using the **Provide feedback** link on the right sidebar of this page.

AWS Console

Note

FHIR export job information is not available on the HealthLake Console. Instead, use the AWS CLI with `ListFHIRExportJobs` to list all FHIR export jobs. For more information, refer to the AWS CLI example on this page.

Code examples for HealthLake using AWS SDKs

The following code examples show how to use HealthLake with an AWS software development kit (SDK).

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios.

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Code examples

- [Basic examples for HealthLake using AWS SDKs](#)
 - [Actions for HealthLake using AWS SDKs](#)
 - [Use CreateFHIRDatastore with an AWS SDK or CLI](#)
 - [Use DeleteFHIRDatastore with an AWS SDK or CLI](#)
 - [Use DescribeFHIRDatastore with an AWS SDK or CLI](#)
 - [Use DescribeFHIRExportJob with an AWS SDK or CLI](#)
 - [Use DescribeFHIRImportJob with an AWS SDK or CLI](#)
 - [Use ListFHIRDatastores with an AWS SDK or CLI](#)
 - [Use ListFHIRExportJobs with an AWS SDK or CLI](#)
 - [Use ListFHIRImportJobs with an AWS SDK or CLI](#)
 - [Use ListTagsForResource with an AWS SDK or CLI](#)
 - [Use StartFHIRExportJob with an AWS SDK or CLI](#)
 - [Use StartFHIRImportJob with an AWS SDK or CLI](#)
 - [Use TagResource with an AWS SDK or CLI](#)
 - [Use UntagResource with an AWS SDK or CLI](#)

Basic examples for HealthLake using AWS SDKs

The following code examples show how to use the basics of AWS HealthLake with AWS SDKs.

Examples

- [Actions for HealthLake using AWS SDKs](#)
 - [Use CreateFHIRDatastore with an AWS SDK or CLI](#)
 - [Use DeleteFHIRDatastore with an AWS SDK or CLI](#)
 - [Use DescribeFHIRDatastore with an AWS SDK or CLI](#)
 - [Use DescribeFHIRExportJob with an AWS SDK or CLI](#)
 - [Use DescribeFHIRImportJob with an AWS SDK or CLI](#)
 - [Use ListFHIRDatastores with an AWS SDK or CLI](#)
 - [Use ListFHIRExportJobs with an AWS SDK or CLI](#)
 - [Use ListFHIRImportJobs with an AWS SDK or CLI](#)
 - [Use ListTagsForResource with an AWS SDK or CLI](#)
 - [Use StartFHIRExportJob with an AWS SDK or CLI](#)
 - [Use StartFHIRImportJob with an AWS SDK or CLI](#)
 - [Use TagResource with an AWS SDK or CLI](#)
 - [Use UntagResource with an AWS SDK or CLI](#)

Actions for HealthLake using AWS SDKs

The following code examples demonstrate how to perform individual HealthLake actions with AWS SDKs. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the [AWS HealthLake API Reference](#).

Examples

- [Use CreateFHIRDatastore with an AWS SDK or CLI](#)
- [Use DeleteFHIRDatastore with an AWS SDK or CLI](#)
- [Use DescribeFHIRDatastore with an AWS SDK or CLI](#)
- [Use DescribeFHIRExportJob with an AWS SDK or CLI](#)
- [Use DescribeFHIRImportJob with an AWS SDK or CLI](#)
- [Use ListFHIRDatastores with an AWS SDK or CLI](#)
- [Use ListFHIRExportJobs with an AWS SDK or CLI](#)
- [Use ListFHIRImportJobs with an AWS SDK or CLI](#)

- [Use ListTagsForResource with an AWS SDK or CLI](#)
- [Use StartFHIRExportJob with an AWS SDK or CLI](#)
- [Use StartFHIRImportJob with an AWS SDK or CLI](#)
- [Use TagResource with an AWS SDK or CLI](#)
- [Use UntagResource with an AWS SDK or CLI](#)

Use CreateFHIRDatastore with an AWS SDK or CLI

The following code examples show how to use CreateFHIRDatastore.

CLI

AWS CLI

Example 1: Create a SigV4-enabled HealthLake data store

The following create-fhir-datastore example demonstrates how to create a new data store in AWS HealthLake.

```
aws healthlake create-fhir-datastore \
  --datastore-type-version R4 \
  --datastore-name "FhirTestDatastore"
```

Output:

```
{
  "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
(Data store ID)/r4/",
  "DatastoreArn": "arn:aws:healthlake:us-east-1:(AWS Account ID):datastore/
(Data store ID)",
  "DatastoreStatus": "CREATING",
  "DatastoreId": "(Data store ID)"
}
```

Example 2: Create a SMART on FHIR-enabled HealthLake data store

The following create-fhir-datastore example demonstrates how to create a new SMART on FHIR-enabled data store in AWS HealthLake.

```
aws healthlake create-fhir-datastore \
```

```
--datastore-name "your-data-store-name" \
--datastore-type-version R4 \
--preload-data-config PreloadDataType="SYNTHEA" \
--sse-configuration '{ "KmsEncryptionConfig": { "CmkType":
"CUSTOMER_MANAGED_KMS_KEY", "KmsKeyId": "arn:aws:kms:us-east-1:your-account-
id:key/your-key-id" } }' \
--identity-provider-configuration file://
identity_provider_configuration.json
```

Contents of `identity_provider_configuration.json`:

```
{
  "AuthorizationStrategy": "SMART_ON_FHIR_V1",
  "FineGrainedAuthorizationEnabled": true,
  "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-id:function:your-
lambda-name",
  "Metadata": "{ \"issuer\": \"https://ehr.example.com\", \"jwks_uri\":
\"https://ehr.example.com/.well-known/jwks.json\", \"authorization_endpoint
\": \"https://ehr.example.com/auth/authorize\", \"token_endpoint\": \"https://
ehr.token.com/auth/token\", \"token_endpoint_auth_methods_supported\":
[\"client_secret_basic\", \"foo\"], \"grant_types_supported\": [\"client_credentia
l\", \"foo\"], \"registration_endpoint\": \"https://ehr.example.com/auth/
register\", \"scopes_supported\": [\"openid\", \"profile\", \"launch\"],
\"response_types_supported\": [\"code\"], \"management_endpoint\": \"https://
ehr.example.com/user/manage\", \"introspection_endpoint\": \"https://
ehr.example.com/user/introspect\", \"revocation_endpoint\": \"https://
ehr.example.com/user/revoke\", \"code_challenge_methods_supported\": [\"S256\"],
\"capabilities\": [\"launch-ehr\", \"sso-openid-connect\", \"client-public\"]}"
}
```

Output:

```
{
  "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
(Data store ID)/r4/",
  "DatastoreArn": "arn:aws:healthlake:us-east-1:(AWS Account ID):datastore/
(Data store ID)",
  "DatastoreStatus": "CREATING",
  "DatastoreId": "(Data store ID)"
}
```

- For API details, see [CreateFHIRDatastore](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def create_fhir_datastore(
    self,
    datastore_name: str,
    sse_configuration: dict[str, any] = None,
    identity_provider_configuration: dict[str, any] = None,
) -> dict[str, str]:
    """
    Creates a new HealthLake data store.
    When creating a SMART on FHIR data store, the following parameters are
    required:
        - sse_configuration: The server-side encryption configuration for a SMART
        on FHIR-enabled data store.
        - identity_provider_configuration: The identity provider configuration
        for a SMART on FHIR-enabled data store.

    :param datastore_name: The name of the data store.
    :param sse_configuration: The server-side encryption configuration for a
    SMART on FHIR-enabled data store.
    :param identity_provider_configuration: The identity provider
    configuration for a SMART on FHIR-enabled data store.
    :return: A dictionary containing the data store information.
    """
    try:
        parameters = {"DatastoreName": datastore_name,
            "DatastoreTypeVersion": "R4"}
        if (
            sse_configuration is not None

```

```

        and identity_provider_configuration is not None
    ):
        # Creating a SMART on FHIR-enabled data store
        parameters["SseConfiguration"] = sse_configuration
        parameters[
            "IdentityProviderConfiguration"
        ] = identity_provider_configuration

    response =
self.health_lake_client.create_fhir_datastore(**parameters)
    return response
except ClientError as err:
    logger.exception(
        "Couldn't create data store %s. Here's why %s",
        datastore_name,
        err.response["Error"]["Message"],
    )
    raise

```

The following code shows an example of parameters for a SMART on FHIR-enabled HealthLake data store.

```

sse_configuration = {
    "KmsEncryptionConfig": {"CmkType": "AWS_OWNED_KMS_KEY"}
}
# TODO: Update the metadata to match your environment.
metadata = {
    "issuer": "https://ehr.example.com",
    "jwks_uri": "https://ehr.example.com/.well-known/jwks.json",
    "authorization_endpoint": "https://ehr.example.com/auth/
authorize",
    "token_endpoint": "https://ehr.token.com/auth/token",
    "token_endpoint_auth_methods_supported": [
        "client_secret_basic",
        "foo",
    ],
    "grant_types_supported": ["client_credential", "foo"],
    "registration_endpoint": "https://ehr.example.com/auth/register",
    "scopes_supported": ["openId", "profile", "launch"],
    "response_types_supported": ["code"],
    "management_endpoint": "https://ehr.example.com/user/manage",

```



```

        "introspection_endpoint": "https://ehr.example.com/user/
introspect",
        "revocation_endpoint": "https://ehr.example.com/user/revoke",
        "code_challenge_methods_supported": ["S256"],
        "capabilities": [
            "launch-ehr",
            "sso-openid-connect",
            "client-public",
        ],
    }
    # TODO: Update the IdpLambdaArn.
    identity_provider_configuration = {
        "AuthorizationStrategy": "SMART_ON_FHIR_V1",
        "FineGrainedAuthorizationEnabled": True,
        "IdpLambdaArn": "arn:aws:lambda:your-region:your-account-
id:function:your-lambda-name",
        "Metadata": json.dumps(metadata),
    }
    data_store = self.create_fhir_datastore(
        datastore_name, sse_configuration,
        identity_provider_configuration
    )

```

- For API details, see [CreateFHIRDatastore](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use DeleteFHIRDatastore with an AWS SDK or CLI

The following code examples show how to use DeleteFHIRDatastore.

CLI

AWS CLI

To delete a FHIR data store

The following `delete-fhir-datastore` example demonstrates how to delete a data store and all of its contents in AWS HealthLake.

```
aws healthlake delete-fhir-datastore \
  --datastore-id (Data store ID)
```

Output:

```
{
  "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
(Data store ID)/r4/",
  "DatastoreArn": "arn:aws:healthlake:us-east-1:(AWS Account ID):datastore/
(Data store ID)",
  "DatastoreStatus": "DELETING",
  "DatastoreId": "(Data store ID)"
}
```

- For API details, see [DeleteFHIRDatastore](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)
```

```
def delete_fhir_datastore(self, datastore_id: str) -> None:
    """
    Deletes a HealthLake data store.
    :param datastore_id: The data store ID.
    """
    try:

self.health_lake_client.delete_fhir_datastore(DatastoreId=datastore_id)
    except ClientError as err:
        logger.exception(
            "Couldn't delete data store with ID %s. Here's why %s",
            datastore_id,
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [DeleteFHIRDatastore](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use DescribeFHIRDatastore with an AWS SDK or CLI

The following code examples show how to use DescribeFHIRDatastore.

CLI

AWS CLI

To describe a FHIR data store

The following describe-fhir-datastore example demonstrates how to find the properties of a data store in AWS HealthLake.

```
aws healthlake describe-fhir-datastore \
  --datastore-id "1f2f459836ac6c513ce899f9e4f66a59"
```

Output:

```
{
  "DatastoreProperties": {
    "PreloadDataConfig": {
      "PreloadDataType": "SYNTHEA"
    },
    "SseConfiguration": {
      "KmsEncryptionConfig": {
        "CmkType": "CUSTOMER_MANAGED_KMS_KEY",
        "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
      }
    },
    "DatastoreName": "Demo",
    "DatastoreArn": "arn:aws:healthlake:us-east-1:<AWS Account ID>:datastore/<Data store ID>",
    "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/<Data store ID>/r4/",
    "DatastoreStatus": "ACTIVE",
    "DatastoreTypeVersion": "R4",
    "CreatedAt": 1603761064.881,
    "DatastoreId": "<Data store ID>",
    "IdentityProviderConfiguration": {
      "AuthorizationStrategy": "AWS_AUTH",
      "FineGrainedAuthorizationEnabled": false
    }
  }
}
```

- For API details, see [DescribeFHIRDatastore](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
```

```

        """
        Creates a HealthLakeWrapper instance with a default AWS HealthLake
        client.

        :return: An instance of HealthLakeWrapper initialized with the default
        HealthLake client.
        """
        health_lake_client = boto3.client("healthlake")
        return cls(health_lake_client)

def describe_fhir_datastore(self, datastore_id: str) -> dict[str, any]:
    """
    Describes a HealthLake data store.
    :param datastore_id: The data store ID.
    :return: The data store description.
    """
    try:
        response = self.health_lake_client.describe_fhir_datastore(
            DatastoreId=datastore_id
        )
        return response["DatastoreProperties"]
    except ClientError as err:
        logger.exception(
            "Couldn't describe data store with ID %s. Here's why %s",
            datastore_id,
            err.response["Error"]["Message"],
        )
        raise

```

- For API details, see [DescribeFHIRDatastore](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use DescribeFHIRExportJob with an AWS SDK or CLI

The following code examples show how to use DescribeFHIRExportJob.

CLI

AWS CLI

To describe a FHIR export job

The following describe-fhir-export-job example shows how to find the properties of a FHIR export job in AWS HealthLake.

```
aws healthlake describe-fhir-export-job \
  --datastore-id (Data store ID) \
  --job-id 9b9a51943afaedd0a8c0c26c49135a31
```

Output:

```
{
  "ExportJobProperties": {
    "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role Name)",
    "JobStatus": "IN_PROGRESS",
    "JobId": "9009813e9d69ba7cf79bcb3468780f16",
    "SubmitTime": "2024-11-20T11:31:46.672000-05:00",
    "EndTime": "2024-11-20T11:34:01.636000-05:00",
    "OutputDataConfig": {
      "S3Configuration": {
        "S3Uri": "s3://(Bucket Name)/(Prefix Name)/",
        "KmsKeyId": "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-
b56c-4216-a250-f4c43ef46e83"
      }
    },
    "DatastoreId": "(Data store ID)"
  }
}
```

- For API details, see [DescribeFHIRExportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def describe_fhir_export_job(
    self, datastore_id: str, job_id: str
) -> dict[str, any]:
    """
    Describes a HealthLake export job.
    :param datastore_id: The data store ID.
    :param job_id: The export job ID.
    :return: The export job description.
    """
    try:
        response = self.health_lake_client.describe_fhir_export_job(
            DatastoreId=datastore_id, JobId=job_id
        )
        return response["ExportJobProperties"]
    except ClientError as err:
        logger.exception(
            "Couldn't describe export job with ID %s. Here's why %s",
            job_id,
            err.response["Error"]["Message"],
        )
        raise
```

- For API details, see [DescribeFHIRExportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

 Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use DescribeFHIRImportJob with an AWS SDK or CLI

The following code examples show how to use DescribeFHIRImportJob.

CLI

AWS CLI

To describe a FHIR import job

The following describe-fhir-import-job example shows how to learn the properties of a FHIR import job using AWS HealthLake.

```
aws healthlake describe-fhir-import-job \  
  --datastore-id (Data store ID) \  
  --job-id c145fbb27b192af392f8ce6e7838e34f
```

Output:

```
{  
  "ImportJobProperties": {  
    "InputDataConfig": {  
      "S3Uri": "s3://(Bucket Name)/(Prefix Name)/"  
      { "arrayitem2": 2 }  
    },  
    "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role Name)",  
    "JobStatus": "COMPLETED",  
    "JobId": "c145fbb27b192af392f8ce6e7838e34f",  
    "SubmitTime": 1606272542.161,  
  },  
}
```



```

    "EndTime": 1606272609.497,
    "DatastoreId": "(Data store ID)"
  }
}

```

- For API details, see [DescribeFHIRImportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def describe_fhir_import_job(
    self, datastore_id: str, job_id: str
) -> dict[str, any]:
    """
    Describes a HealthLake import job.
    :param datastore_id: The data store ID.
    :param job_id: The import job ID.
    :return: The import job description.
    """
    try:
        response = self.health_lake_client.describe_fhir_import_job(
            DatastoreId=datastore_id, JobId=job_id
        )
        return response["ImportJobProperties"]
    except ClientError as err:
        logger.exception(
            "Couldn't describe import job with ID %s. Here's why %s",
            job_id,

```

```
        err.response["Error"]["Message"],
    )
    raise
```

- For API details, see [DescribeFHIRImportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use ListFHIRDatastores with an AWS SDK or CLI

The following code examples show how to use ListFHIRDatastores.

CLI

AWS CLI

To list FHIR data stores

The following list-fhir-datastores example shows to how to use the command and how users can filter results based on data store status in AWS HealthLake.

```
aws healthlake list-fhir-datastores \
    --filter DatastoreStatus=ACTIVE
```

Output:

```
{
  "DatastorePropertiesList": [
    {
      "PreloadDataConfig": {
        "PreloadDataType": "SYNTHEA"
```

```

    },
    "SseConfiguration": {
        "KmsEncryptionConfig": {
            "CmkType": "CUSTOMER_MANAGED_KMS_KEY",
            "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
        }
    },
    "DatastoreName": "Demo",
    "DatastoreArn": "arn:aws:healthlake:us-east-1:<AWS Account ID>:datastore/
<Data store ID>",
    "DatastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/
datastore/<Data store ID>/r4/",
    "DatastoreStatus": "ACTIVE",
    "DatastoreTypeVersion": "R4",
    "CreatedAt": 1603761064.881,
    "DatastoreId": "<Data store ID>",
    "IdentityProviderConfiguration": {
        "AuthorizationStrategy": "AWS_AUTH",
        "FineGrainedAuthorizationEnabled": false
    }
}
]
}

```

- For API details, see [ListFHIRDatastores](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

```

```

def list_fhir_datastores(self) -> list[dict[str, any]]:
    """
    Lists all HealthLake data stores.
    :return: A list of data store descriptions.
    """
    try:
        next_token = None
        datastores = []

        # Loop through paginated results.
        while True:
            parameters = {}
            if next_token is not None:
                parameters["NextToken"] = next_token
            response =
self.health_lake_client.list_fhir_datastores(**parameters)
            datastores.extend(response["DatastorePropertiesList"])
            if "NextToken" in response:
                next_token = response["NextToken"]
            else:
                break

        return datastores
    except ClientError as err:
        logger.exception(
            "Couldn't list data stores. Here's why %s", err.response["Error"]
["Message"]
        )
        raise

```

- For API details, see [ListFHIRDatastores](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use ListFHIRExportJobs with an AWS SDK or CLI

The following code examples show how to use ListFHIRExportJobs.

CLI

AWS CLI

To list all FHIR export jobs

The following list-fhir-export-jobs example shows how to use the command to view a list of export jobs associated with an account.

```
aws healthlake list-fhir-export-jobs \
  --datastore-id (Data store ID) \
  --submitted-before (DATE like 2024-10-13T19:00:00Z) \
  --submitted-after (DATE like 2020-10-13T19:00:00Z ) \
  --job-name "FHIR-EXPORT" \
  --job-status SUBMITTED \
  --max-results (Integer between 1 and 500)
```

Output:

```
{
  "ExportJobPropertiesList": [
    {
      "ExportJobProperties": {
        "OutputDataConfig": {
          "S3Uri": "s3://(Bucket Name)/(Prefix Name)/",
          "S3Configuration": {
            "S3Uri": "s3://(Bucket Name)/(Prefix Name)/",
            "KmsKeyId": "(KmsKey Id)"
          }
        },
        "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role Name)",
        "JobStatus": "COMPLETED",
        "JobId": "c145fbb27b192af392f8ce6e7838e34f",
        "JobName": "FHIR-EXPORT",
```

```

        "SubmitTime": "2024-11-20T11:31:46.672000-05:00",
        "EndTime": "2024-11-20T11:34:01.636000-05:00",
        "DatastoreId": "(Data store ID)"
    }
}
]
}

```

- For API details, see [ListFHIRExportJobs](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def list_fhir_export_jobs(
    self,
    datastore_id: str,
    job_name: str = None,
    job_status: str = None,
    submitted_before: datetime = None,
    submitted_after: datetime = None,
) -> list[dict[str, any]]:
    """
    Lists HealthLake export jobs satisfying the conditions.
    :param datastore_id: The data store ID.
    :param job_name: The export job name.
    :param job_status: The export job status.
    :param submitted_before: The export job submitted before the specified
    date.
    """

```

```

        :param submitted_after: The export job submitted after the specified
        date.
        :return: A list of export jobs.
        """
        try:
            parameters = {"DatastoreId": datastore_id}
            if job_name is not None:
                parameters["JobName"] = job_name
            if job_status is not None:
                parameters["JobStatus"] = job_status
            if submitted_before is not None:
                parameters["SubmittedBefore"] = submitted_before
            if submitted_after is not None:
                parameters["SubmittedAfter"] = submitted_after
            next_token = None
            jobs = []
            # Loop through paginated results.
            while True:
                if next_token is not None:
                    parameters["NextToken"] = next_token
                response =
self.health_lake_client.list_fhir_export_jobs(**parameters)
                jobs.extend(response["ExportJobPropertiesList"])
                if "NextToken" in response:
                    next_token = response["NextToken"]
                else:
                    break
            return jobs
        except ClientError as err:
            logger.exception(
                "Couldn't list export jobs. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise

```

- For API details, see [ListFHIRExportJobs](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use ListFHIRImportJobs with an AWS SDK or CLI

The following code examples show how to use ListFHIRImportJobs.

CLI

AWS CLI

To list all FHIR import jobs

The following list-fhir-import-jobs example shows how to use the command to view a list of all import jobs associated with an account.

```
aws healthlake list-fhir-import-jobs \
  --datastore-id (Data store ID) \
  --submitted-before (DATE like 2024-10-13T19:00:00Z) \
  --submitted-after (DATE like 2020-10-13T19:00:00Z ) \
  --job-name "FHIR-IMPORT" \
  --job-status SUBMITTED \
  --max-results (Integer between 1 and 500)
```

Output:

```
{
  "ImportJobPropertiesList": [
    {
      "JobId": "c0fd dbf76f238297632d4aebdbfc9ddf",
      "JobStatus": "COMPLETED",
      "SubmitTime": "2024-11-20T10:08:46.813000-05:00",
      "EndTime": "2024-11-20T10:10:09.093000-05:00",
      "DatastoreId": "(Data store ID)",
```



```

        "InputDataConfig": {
            "S3Uri": "s3://(Bucket Name)/(Prefix Name)/"
        },
        "JobOutputDataConfig": {
            "S3Configuration": {
                "S3Uri": "s3://(Bucket Name)/
import/6407b9ae4c2def3cb6f1a46a0c599ec0-FHIR_IMPORT-
c0fddb76f238297632d4aebdbfc9ddf/",
                "KmsKeyId": "arn:aws:kms:us-east-1:123456789012:key/b7f645cb-
e564-4981-8672-9e012d1ff1a0"
            }
        },
        "JobProgressReport": {
            "TotalNumberOfScannedFiles": 1,
            "TotalSizeOfScannedFilesInMB": 0.001798,
            "TotalNumberOfImportedFiles": 1,
            "TotalNumberOfResourcesScanned": 1,
            "TotalNumberOfResourcesImported": 1,
            "TotalNumberOfResourcesWithCustomerError": 0,
            "TotalNumberOfFilesReadWithCustomerError": 0,
            "Throughput": 0.0
        },
        "DataAccessRoleArn": "arn:aws:iam::(AWS Account ID):role/(Role Name)"
    }
}

```

- For API details, see [ListFHIRImportJobs](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """

```

```

health_lake_client = boto3.client("healthlake")
return cls(health_lake_client)

def list_fhir_import_jobs(
    self,
    datastore_id: str,
    job_name: str = None,
    job_status: str = None,
    submitted_before: datetime = None,
    submitted_after: datetime = None,
) -> list[dict[str, any]]:
    """
    Lists HealthLake import jobs satisfying the conditions.
    :param datastore_id: The data store ID.
    :param job_name: The import job name.
    :param job_status: The import job status.
    :param submitted_before: The import job submitted before the specified
date.
    :param submitted_after: The import job submitted after the specified
date.
    :return: A list of import jobs.
    """
    try:
        parameters = {"DatastoreId": datastore_id}
        if job_name is not None:
            parameters["JobName"] = job_name
        if job_status is not None:
            parameters["JobStatus"] = job_status
        if submitted_before is not None:
            parameters["SubmittedBefore"] = submitted_before
        if submitted_after is not None:
            parameters["SubmittedAfter"] = submitted_after
        next_token = None
        jobs = []
        # Loop through paginated results.
        while True:
            if next_token is not None:
                parameters["NextToken"] = next_token
            response =
self.health_lake_client.list_fhir_import_jobs(**parameters)
            jobs.extend(response["ImportJobPropertiesList"])
            if "NextToken" in response:
                next_token = response["NextToken"]

```

```
        else:
            break
    return jobs
except ClientError as err:
    logger.exception(
        "Couldn't list import jobs. Here's why %s",
        err.response["Error"]["Message"],
    )
    raise
```

- For API details, see [ListFHIRImportJobs](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use ListTagsForResource with an AWS SDK or CLI

The following code examples show how to use ListTagsForResource.

CLI

AWS CLI

To list tags for a data store

The following list-tags-for-resource example lists the tags associated with the specified data store.:

```
aws healthlake list-tags-for-resource \
    --resource-arn "arn:aws:healthlake:us-east-1:123456789012:datastore/
    fhir/0725c83f4307f263e16fd56b6d8ebdbe"
```

Output:

```
{
  "tags": {
    "key": "value",
    "key1": "value1"
  }
}
```

- For API details, see [ListTagsForResource](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def list_tags_for_resource(self, resource_arn: str) -> dict[str, str]:
    """
    Lists the tags for a HealthLake resource.
    :param resource_arn: The resource ARN.
    :return: The tags for the resource.
    """
    try:
        response = self.health_lake_client.list_tags_for_resource(
            ResourceARN=resource_arn
        )
        return response["Tags"]
    except ClientError as err:
        logger.exception(
            "Couldn't list tags for resource %s. Here's why %s",
            resource_arn,
            err.response["Error"]["Message"],
        )
```

```
)
raise
```

- For API details, see [ListTagsForResource](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use StartFHIRExportJob with an AWS SDK or CLI

The following code examples show how to use StartFHIRExportJob.

CLI

AWS CLI

To start a FHIR export job

The following start-fhir-export-job example shows how to start a FHIR export job using AWS HealthLake.

```
aws healthlake start-fhir-export-job \
  --output-data-config '{"S3Configuration": {"S3Uri": "s3://(Bucket Name)/
(Prefix Name)/", "KmsKeyId": "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-
b56c-4216-a250-f4c43ef46e83"}}' \
  --datastore-id (Data store ID) \
  --data-access-role-arn arn:aws:iam::(AWS Account ID):role/(Role Name)
```

Output:

```
{
  "DatastoreId": "(Data store ID)",
```

```

    "JobStatus": "SUBMITTED",
    "JobId": "9b9a51943afaedd0a8c0c26c49135a31"
}

```

- For API details, see [StartFHIRExportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```

@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def start_fhir_export_job(
    self,
    job_name: str,
    datastore_id: str,
    output_s3_uri: str,
    kms_key_id: str,
    data_access_role_arn: str,
) -> dict[str, str]:
    """
    Starts a HealthLake export job.
    :param job_name: The export job name.
    :param datastore_id: The data store ID.
    :param output_s3_uri: The output S3 URI.
    :param kms_key_id: The KMS key ID associated with the output S3 bucket.
    :param data_access_role_arn: The data access role ARN.
    :return: The export job.
    """
    try:
        response = self.health_lake_client.start_fhir_export_job(

```

```
        OutputDataConfig={
            "S3Configuration": {"S3Uri": output_s3_uri, "KmsKeyId":
kms_key_id}
        },
        DataAccessRoleArn=data_access_role_arn,
        DatastoreId=datastore_id,
        JobName=job_name,
    )

    return response
except ClientError as err:
    logger.exception(
        "Couldn't start export job. Here's why %s",
        err.response["Error"]["Message"],
    )
    raise
```

- For API details, see [StartFHIRExportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use StartFHIRImportJob with an AWS SDK or CLI

The following code examples show how to use StartFHIRImportJob.

CLI

AWS CLI

To start a FHIR import job

The following `start-fhir-import-job` example shows how to start a FHIR import job using AWS HealthLake.

```
aws healthlake start-fhir-import-job \
  --input-data-config S3Uri="s3://(Bucket Name)/(Prefix Name)/" \
  --job-output-data-config '{"S3Configuration": {"S3Uri": "s3://(Bucket Name)/(Prefix Name)/", "KmsKeyId": "arn:aws:kms:us-east-1:012345678910:key/d330e7fc-b56c-4216-a250-f4c43ef46e83"}}' \
  --datastore-id (Data store ID) \
  --data-access-role-arn "arn:aws:iam::(AWS Account ID):role/(Role Name)"
```

Output:

```
{
  "DatastoreId": "(Data store ID)",
  "JobStatus": "SUBMITTED",
  "JobId": "c145fbb27b192af392f8ce6e7838e34f"
}
```

- For API details, see [StartFHIRImportJob](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def start_fhir_import_job(
    self,
    job_name: str,
```



```

        datastore_id: str,
        input_s3_uri: str,
        job_output_s3_uri: str,
        kms_key_id: str,
        data_access_role_arn: str,
    ) -> dict[str, str]:
        """
        Starts a HealthLake import job.
        :param job_name: The import job name.
        :param datastore_id: The data store ID.
        :param input_s3_uri: The input S3 URI.
        :param job_output_s3_uri: The job output S3 URI.
        :param kms_key_id: The KMS key ID associated with the output S3 bucket.
        :param data_access_role_arn: The data access role ARN.
        :return: The import job.
        """
        try:
            response = self.health_lake_client.start_fhir_import_job(
                JobName=job_name,
                InputDataConfig={"S3Uri": input_s3_uri},
                JobOutputDataConfig={
                    "S3Configuration": {
                        "S3Uri": job_output_s3_uri,
                        "KmsKeyId": kms_key_id,
                    }
                },
                DataAccessRoleArn=data_access_role_arn,
                DatastoreId=datastore_id,
            )
            return response
        except ClientError as err:
            logger.exception(
                "Couldn't start import job. Here's why %s",
                err.response["Error"]["Message"],
            )
            raise

```

- For API details, see [StartFHIRImportJob](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use TagResource with an AWS SDK or CLI

The following code examples show how to use TagResource.

CLI

AWS CLI

To add a tag to data store

The following tag-resource example shows how to add a tag to a data store.

```
aws healthlake tag-resource \  
  --resource-arn "arn:aws:healthlake:us-east-1:123456789012:datastore/  
fhir/0725c83f4307f263e16fd56b6d8ebdbe" \  
  --tags '[{"Key": "key1", "Value": "value1"}]'
```

This command produces no output.

- For API details, see [TagResource](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod  
def from_client(cls) -> "HealthLakeWrapper":  
    """  
    Creates a HealthLakeWrapper instance with a default AWS HealthLake  
    client.
```

```

        :return: An instance of HealthLakeWrapper initialized with the default
        HealthLake client.
        """
        health_lake_client = boto3.client("healthlake")
        return cls(health_lake_client)

    def tag_resource(self, resource_arn: str, tags: list[dict[str, str]]) ->
    None:
        """
        Tags a HealthLake resource.
        :param resource_arn: The resource ARN.
        :param tags: The tags to add to the resource.
        """
        try:
            self.health_lake_client.tag_resource(ResourceARN=resource_arn,
            Tags=tags)
        except ClientError as err:
            logger.exception(
                "Couldn't tag resource %s. Here's why %s",
                resource_arn,
                err.response["Error"]["Message"],
            )
            raise

```

- For API details, see [TagResource](#) in *AWS SDK for Python (Boto3) API Reference*.

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Use UntagResource with an AWS SDK or CLI

The following code examples show how to use UntagResource.

CLI

AWS CLI

To remove tags from a data store.

The following `untag-resource` example shows how to remove tags from a data store.

```
aws healthlake untag-resource \
    --resource-arn "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
b91723d65c6fdeb1d26543a49d2ed1fa" \
    --tag-keys '["key1"]'
```

This command produces no output.

- For API details, see [UntagResource](#) in *AWS CLI Command Reference*.

Python

SDK for Python (Boto3)

```
@classmethod
def from_client(cls) -> "HealthLakeWrapper":
    """
    Creates a HealthLakeWrapper instance with a default AWS HealthLake
    client.

    :return: An instance of HealthLakeWrapper initialized with the default
    HealthLake client.
    """
    health_lake_client = boto3.client("healthlake")
    return cls(health_lake_client)

def untag_resource(self, resource_arn: str, tag_keys: list[str]) -> None:
    """
    Untags a HealthLake resource.
    :param resource_arn: The resource ARN.
    :param tag_keys: The tag keys to remove from the resource.
    """
    try:
        self.health_lake_client.untag_resource(
```

```
        ResourceARN=resource_arn, TagKeys=tag_keys
    )
except ClientError as err:
    logger.exception(
        "Couldn't untag resource %s. Here's why %s",
        resource_arn,
        err.response["Error"]["Message"],
    )
    raise
```

- For API details, see [UntagResource](#) in *AWS SDK for Python (Boto3) API Reference*.

 Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

For a complete list of AWS SDK developer guides and code examples, see [Using HealthLake with an AWS SDK](#). This topic also includes information about getting started and details about previous SDK versions.

Integrating AWS HealthLake

The following AWS services integrate directly with AWS HealthLake to enable integrated natural language processing, SQL query, and data warehousing.

- *Amazon Comprehend Medical* is a HIPAA eligible natural language processing service (NLP) that uses machine learning libraries to extract meaningful health data from unstructured medical text in HealthLake. For more information, see the [Amazon Comprehend Medical Developer Guide](#).
- *Amazon Athena* is an interactive query service that enables you analyze HealthLake data directly in Amazon Simple Storage Service (Amazon S3) buckets using standard SQL. For more information, see the [Amazon Athena Developer Guide](#).

Topics

- [Integrated natural language processing \(NLP\) for HealthLake](#)
- [Querying HealthLake data with Amazon Athena](#)

Integrated natural language processing (NLP) for HealthLake

AWS HealthLake provides integrated natural language processing (NLP) libraries for parsing, identifying, and mapping unstructured data stored in FHIR [DocumentReference](#) resource types.

Important

Integrated NLP for HealthLake is turned off by default. To have it turned on, submit a support case using [AWS Support Center Console](#). To create your case, log in to your AWS account and choose **Create case**.

HealthLake integrated NLP works by calling the Amazon Comprehend Medical DetectEntities-V2, InferICD10-CM, and InferRxNorm API actions. The actions append their results as an extension to the DocumentReference resource. When the Amazon Comprehend Medical API actions detect traits that are SIGN, SYMPTOM, and/or DIAGNOSIS, they generate a FHIR [Linkage](#) resource. New Condition and Observation resources are created from entities identified with the traits of SIGN, SYMPTOM, or DIAGNOSIS and they are linked to the source document using the Linkage resource.

Note

Although GET requests are supported for FHIR resources generated by HealthLake integrated NLP, FHIR API search functionality is not. To learn more about searching NLP extensions using HealthLake's integration with Athena, see [SQL index and query](#).

Contents

- [HealthLake Integrated NLP libraries](#)
- [Using FHIR REST API interactions](#)
- [Search parameters for HealthLake integrated NLP](#)
- [HealthLake integrated NLP example requests](#)

HealthLake Integrated NLP libraries

HealthLake infers data found in the DocumentReference resource type using Amazon Comprehend Medical libraries. The Amazon Comprehend Medical API operations DetectEntities-V2, InferICD10-CM, and InferRxNorm detect medical conditions as *traits*. Each operation provides different insights.

⚠ Language support

Amazon Comprehend Medical API operations only detect medical entities in English language texts.

- **DetectEntities-V2:** Inspects the clinical text for a variety of medical entities and returns specific information about them, such as entity category, location, and confidence score.
- **InferICD10-CM:** Detects medical conditions in a patient record as entities, and it links those entities to normalized concept identifiers in the ICD-10-CM knowledge base from the CDC's National Center for Health Statistics under authorization by the World Health Organization (WHO).
- **InferRxNorm:** Detects medications as entities listed in a patient record, and it links them to the normalized concept identifiers in the RxNorm database from the National Library of Medicine.

The supported traits for each API operation are SIGN, SYMPTOM, and DIAGNOSIS. If traits are detected, they are added as FHIR-compliant extensions to different locations in your HealthLake data store.

Locations where extensions are added.

- **DocumentReference:** The results from the Amazon Comprehend Medical API operations are added as an extension to each document found within the DocumentReference resource type. Results in the extension are divided into two groups. You can find them in the results based on their URL.
 - `http://healthlake.amazonaws.com/system-generated-resources/`
 - These are resource types that have been created or added to by HealthLake.
 - `http://healthlake.amazonaws.com/aws-cm/`
 - Where the raw output of the Amazon Comprehend Medical API operations is added to your HealthLake data store.
- **Linkage:** This resource type is either added or created as a result of the integrated NLP. A GET request on a specific Linkage returns a list of linked resources. To identify if a Linkage was added by HealthLake, look for the added "tag": [{"display": "SYSTEM_GENERATED"}] key-value pair. To learn more about the FHIR specifications for Linkage, see [Linkage](#) in the FHIR R4 documentation.
- **FHIR resource types generated as a result of the Amazon Comprehend Medical operations.**
 - **Observation:** includes results from the Amazon Comprehend Medical API actions DetectEntities-V2 and InferICD10-CM when the traits are SIGN or SYMPTOM.
 - **Condition:** includes results from the Amazon Comprehend Medical API actions DetectEntities-V2 and InferICD10-CM when the trait is DIAGNOSIS.
 - **MedicationStatement:** includes results from the Amazon Comprehend Medical API actions InferRxNorm.

Using FHIR REST API interactions

By default, traits detected by the Amazon Comprehend Medical API operations are not returned when making a GET request. To see the results of the integrated NLP operations, you must specify a known ID for the following FHIR resource types.

- Linkage

- Observation
- Condition
- MedicationStatement

The results of HealthLake integrated NLP actions outside the DocumentReference resource type are available using a GET request where the specified ID is known to contain results from the Amazon Comprehend Medical API operations.

Search parameters for HealthLake integrated NLP

The following table lists the searchable attributes for HealthLake integrated NLP.

Search parameters for HealthLake NLP

Search parameters	Finds matches for
detectEntities-entity-category	Entity Category within the DetectEntities subextension within the AWS CM Extension
detectEntities-entity-text	Entity Text within the DetectEntities subextension within the AWS CM Extension
detectEntities-entity-type	Entity Type within the DetectEntities subextension within the AWS CM Extension
detectEntities-entity-score	Entity Score within the DetectEntities subextension within the AWS CM Extension
infer-icd10cm-entity-text	Entity Text within the InferICD10CM subextension within the AWS CM Extension
infer-icd10cm-entity-score	Entity Score within the InferICD10CM subextension within the AWS CM Extension
infer-icd10cm-entity-concept-code	Entity Concept Code within the InferICD10CM subextension within the AWS CM Extension
infer-icd10cm-entity-concept-description	Entity Concept Description within the InferICD10CM subextension within the AWS CM Extension

Search parameters	Finds matches for
infer-icd10cm-entity-concept-score	Entity Concept Score within the InferICD10CM subextension within the AWS CM Extension
infer-rxnorm-entity-score	Entity Score within the InferRxNorm subextension within the AWS CM Extension
infer-rxnorm-entity-text	Entity Text within the InferRxNorm subextension within the AWS CM Extension
infer-rxnorm-entity-concept-code	Entity Concept Code within the InferRxNorm subextension within the AWS CM Extension
infer-rxnorm-entity-concept-description	Entity Concept Description within the InferRxNorm subextension within the AWS CM Extension
infer-rxnorm-entity-concept-score	Entity Concept Score within the InferRxNorm subextension within the AWS CM Extension

HealthLake provides a special search to match the criteria where `EntityText` and `EntityCategory` are part of the same entity. The following table describes the special search parameters supported by HealthLake.

Search parameters

Search parameters	Matches returned
detectEntities-entity-text-category	If there is at least one entity in the DetectEntities subextension that matches both the <code>entityText</code> and <code>entityCategory</code> .
detectEntities-entity-type-score	If there is at least one entity in the DetectEntities subextension that matches both the <code>entityType</code> and <code>entityScore</code> .
detectEntities-entity-text-score	If there is at least one entity in the DetectEntities subextension that matches both the <code>entityText</code> and <code>entityScore</code> .

Search parameters	Matches returned
detectEntities-entity-text-type	If there is at least one entity in the DetectEntities subextension that matches both the entityText and entityType.
detectEntities-entity-category-score	If there is at least one entity that matches both the entityCategory and entityScore.
infer-icd10cm-entity-text-concept-code	If there is at least one entity in the InferICD10CM sub-extension that matches the entityText and there is at least one conceptCode for that entity that matches the code.
infer-icd10cm-entity-text-concept-score	If there is at least one entity in the InferICD10CM sub-extension that matches the entityText and there is at least one conceptScore for that entity that matches the score.
infer-icd10cm-entity-concept-description-concept-score	If there is at least one concept within the entity in the InferICD10CM sub-extension that matches the concept description and the conceptScore.
infer-rxnorm-entity-text-concept-code	If there is at least one entity in the InferRxNorm sub-extension that matches the entityText and there is at least one conceptCode for that entity that matches the code.
infer-rxnorm-entity-text-concept-score	If there is at least one entity in the InferRxNorm sub-extension that matches the entityText and there is at least one conceptScore for that entity that matches the score.
infer-rxnorm-entity-concept-description-concept-score	If there is at least one concept within the entity in the InferRxNorm sub-extension that matches the concept description and the conceptScore.

HealthLake integrated NLP example requests

Example 1: Patient record ingested into a HealthLake data store

Following is an example of a clinical note based on of a Patient encounter with a health care professional.

Synthetic data

The text in the following example is synthetic content and does not contain protected health information (PHI).

1991-08-31

Chief Complaint

- Headache
- Sinus Pain
- Nasal Congestion
- Sore Throat
- Pain with Bright Lights
- Nasal Discharge
- Cough

History of Present Illness

Jerónimo599 is a 4 month-old non-hispanic white male.

Social History

Patient has never smoked.

Patient comes from a middle socioeconomic background.

Patient currently has Aetna.

Allergies

No Known Allergies.

Medications

No Active Medications.

Assessment and Plan

```
Patient is presenting with bee venom (substance), mold (organism), house dust
mite (organism), animal dander (substance), grass pollen (substance), tree pollen
(substance), lisinopril, sulfamethoxazole / trimethoprim, fish (substance).
```

```
## Plan
```

```
The patient was prescribed the following medications:
```

- astemizole 10 mg oral tablet
- nda020800 0.3 ml epinephrine 1 mg/ml auto-injector

```
The patient was placed on a careplan:
```

- self-care interventions (procedure)

As a reminder, this information is encoded in base64 format in the DocumentReference resource. When this document is ingested into HealthLake and the Amazon Comprehend Medical API operations are complete, to see the results, you can start with the GET request on the DocumentReference resource type.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/
r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/DocumentReference
```

When the Amazon Comprehend Medical API operations are successful, look for these key-value pairs inside the extension linked to the following "url": "http://healthlake.amazonaws.com/aws-cm/"

```
{
  "url": "http://healthlake.amazonaws.com/aws-cm/status/",
  "valueString": "SUCCESS"
},
{
  "url": "http://healthlake.amazonaws.com/aws-cm/message/",
  "valueString": "The AWS HealthLake integrated medical NLP operation was successful."
}
```

The following tabs show you how the ingested medical record is reported in your HealthLake data store based on the resource type.

DocumentReference

To see the results for a single DocumentReference resource type, make a GET request where the id of a specific resource is provided.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/
r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/DocumentReference/0e938f03-da7f-4178-acd8-
eea9586c46ed
```

When successful, you get a 200 HTTP response code, and the following JSON response (that has been truncated for clarity).

Here is the `http://healthlake.amazonaws.com/system-generated-resources/` portion. You can see that a new Linkage/`e366d29f-2c22-4c19-866e-09603937935a` has been added. You can also see where HealthLake has added inference-based findings to specific Observation and Condition resource types.

To see how these resource types have been amended, choose the related tabs.

```
{
  "extension": [
    {
      "url": "http://healthlake.amazonaws.com/linkage",
      "valueReference": {
        "reference": "Linkage/e366d29f-2c22-4c19-866e-09603937935a"
      }
    },
    {
      "url": "http://healthlake.amazonaws.com/nlp-entity",
      "valueReference": {
        "reference": "Observation/c6e0a3ff-7a17-4d8b-bfd0-d02d7da090c5"
      }
    },
    {
      "url": "http://healthlake.amazonaws.com/nlp-entity",
      "valueReference": {
        "reference": "Condition/0854e1f3-894d-448e-a8d9-3af5b9902baf"
      }
    }
  ],
  "url": "http://healthlake.amazonaws.com/system-generated-resources/"
}
```

Linkage

To see the results for a single Linkage resource type, make a GET request where the ID of a specific resource is provided.

```
GET https://https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/Linkage/e366d29f-2c22-4c19-866e-09603937935a
```

When successful, you get a 200 HTTP response code, and the following truncated JSON response.

The response contains the `item` element. In it, the key-value pair `"type": "source"` indicates the specific `DocumentReference` entry used to modify the `Condition` and `Observations` listed under the `"type": "alternate"` key-value pair.

You also see the `meta` element, and a corresponding key-value pair, `"tag": [{"display": "SYSTEM_GENERATED"}]`, indicating these resources were created by HealthLake.

```
{
  "resourceType": "Linkage",
  "id": "e366d29f-2c22-4c19-866e-09603937935a",
  "active": true,
  "item":
  [
    {
      "type": "alternate",
      "resource": {
        "reference": "Observation/c6e0a3ff-7a17-4d8b-bfd0-d02d7da090c5",
        "type": "Observation"
      }
    },
    {
      "type": "alternate",
      "resource": {
        "reference": "Condition/9d5c1ef6-f822-4faf-b55f-7c70f2a4aa8d",
        "type": "Condition"
      }
    },
    {
      "type": "source",
      "resource": {
        "reference": "DocumentReference/0e938f03-da7f-4178-acd8-eea9586c46ed",
        "type": "DocumentReference"
      }
    }
  ],
  "meta": {
```

```

    "lastUpdated": "2022-10-21T19:38:31.327Z",
    "tag": [{
      "display": "SYSTEM_GENERATED"
    }]
  }
}

```

Resource type: Observation

To see the results for a single Observation resource type, make a GET request where the ID of a specific resource is provided.

```

GET https://healthlake.region.amazonaws.com/
datastore/datastoreId/r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/
Observation/e366d29f-2c22-4c19-866e-09603937935a

```

The results of the Amazon Comprehend Medical API operations are amended to the following elements: code, meta, and modifierExtension.

code

An element of type CodeableConcept. To learn more, see [CodeableConcept](#) in the FHIR R4 documentation.

HealthLake appends the following three key-value pairs.

- "system": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/": Where the URL refers to a specific Amazon Comprehend Medical API operation. In this case, InferICD10CM.
- "code": "A52.06": Where A52.06 is the ICD-10-CM code that identifies the concept found in the knowledge base from the Centers for Disease Control.
- "display": "Other syphilitic heart involvement": Where "Other syphilitic heart involvement" is the long description of the ICD-10-CM code in the ontology.

The following truncated JSON response contains only the code element.

```

"code": {
  "coding":
  [

```



```
{
  "system": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/",
  "code": "A52.06",
  "display": "Other syphilitic heart involvement"
},
{
  "text": "Other syphilitic heart involvement"
}
```

To understand the model's confidence that the assigned ICD-10-CM code is correct, use the `modifierExtension` element.

meta

The `meta` element contains metadata that indicates whether the code element contains details that have been added by the Amazon Comprehend Medical API operations.

The following truncated JSON response contains only the `meta` element.

```
"meta": {
  "lastUpdated": "2022-10-21T19:38:30.879Z",
  "tag": [{
    "display": "SYSTEM_GENERATED"
  }]
}
```

modifierExtension

The `modifierExtension` element contains more details about the level of confidence of the assigned codes found in the code element. It also has key-value pairs that provide a link back to the original `DocumentReference` used to generate the results and the related `Linkage` resource type.

For each coding element added, you will see an `entity-score` and an `entity-Concept-Score` added to the `modifierExtension`. For each value in the key-value pair, you see a score. For `entity-score`, this score is the level of confidence that Amazon Comprehend Medical has in the accuracy of the detection. For `entity-Concept-Score`, this score is the level of confidence that Amazon Comprehend Medical has that the entity is accurately linked to an ICD-10-CM concept.

The following truncated JSON response contains only the `modifierExtension` element.

```

"modifierExtension": [{
  "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-score",
  "valueDecimal": 0.45005733
},
{
  "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-Concept-Score",
  "valueDecimal": 0.1111792
},
{
  "url": "http://healthlake.amazonaws.com/system-generated-linkage",
  "valueReference": {
    "reference": "Linkage/e366d29f-2c22-4c19-866e-09603937935a"
  }
},
{
  "url": "http://healthlake.amazonaws.com/source-document-reference",
  "valueReference": {
    "reference": "DocumentReference/0e938f03-da7f-4178-acd8-eea9586c46ed"
  }
}
]

```

Full JSON Response

```

{
  "subject": {
    "reference": "Patient/0679b7b7-937d-488a-b48d-6315b8e7003b"
  },
  "resourceType": "Observation",
  "status": "unknown",
  "code": {
    "coding": [{
      "system": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/",
      "code": "A52.06",
      "display": "Other syphilitic heart involvement"
    }],
    "text": "Other syphilitic heart involvement"
  },
  "meta": {
    "lastUpdated": "2022-10-21T19:38:30.879Z",
    "tag": [{

```

```

    "display": "SYSTEM_GENERATED"
  }]
},
"modifierExtension": [{
  "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-score",
  "valueDecimal": 0.45005733
},
{
  "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-Concept-Score",
  "valueDecimal": 0.1111792
},
{
  "url": "http://healthlake.amazonaws.com/system-generated-linkage",
  "valueReference": {
    "reference": "Linkage/e366d29f-2c22-4c19-866e-09603937935a"
  }
},
{
  "url": "http://healthlake.amazonaws.com/source-document-reference",
  "valueReference": {
    "reference": "DocumentReference/0e938f03-da7f-4178-acd8-eea9586c46ed"
  }
}
],
"id": "7e88c7c5-21a5-4dd7-8fc2-a02474fba583"
}

```

Condition

To see the results for a single Condition resource type, make a GET request where the ID of a specific resource is provided.

```

GET https://healthlake.region.amazonaws.com/datastore/datastoreId/
r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/Condition/b06d343d-
ddb8-4f36-82cb-853fcd434dfd

```

The results of the Amazon Comprehend Medical API operations are amended to the following elements: code, meta, and modifierExtension.

code

An element of type `CodeableConcept`. To learn more, see [CodeableConcept](#) in the FHIR R4 documentation.

HealthLake appends the following three key-value pairs.

- `"system": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/"`: Where the URL refers to a specific Amazon Comprehend Medical API operation. In this case, `InferICD10CM`.
- `"code": "I70.0"`: Where `A52.06` is the ICD-10-CM code that identifies the concept found in the knowledge base from the Centers for Disease Control.
- `"display": "Atherosclerosis of aorta"`: Where `"Other syphilitic heart involvement"` is the long description of the ICD-10-CM code in the ontology.

The following truncated JSON response contains only the code element.

```
"code": {
  "coding":
  [
    {
      "system": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/",
      "code": "I70.0",
      "display": "Atherosclerosis of aorta"
    }
  ],
  "text": "Atherosclerosis of aorta"
}
```

To understand the model's confidence that the assigned ICD-10-CM code is correct, use the `modifierExtension` element.

meta

The meta element contains metadata that indicates whether the code element contains details that have been added by the Amazon Comprehend Medical API operations.

The following truncated JSON response contains only the meta element.

```
"meta": {
  "lastUpdated": "2022-10-21T19:38:30.877Z",
  "tag": [{
    "display": "SYSTEM_GENERATED"
```

```
    }  
  }  
}
```

modifierExtension

The `modifierExtension` element contains more details about the level of confidence of the assigned codes found in the code element. It also has key-value pairs that provide a link back to the original `DocumentReference` used to generate the results and the related Linkage resource type.

For each coding element added, you will see an `entity-score` and an `entity-Concept-Score` added to the `modifierExtension`. For each value in the key-value pair, you see a score. For `entity-score`, this score is the level of confidence that Amazon Comprehend Medical has in the accuracy of the detection. For `entity-Concept-Score`, this score is the level of confidence that Amazon Comprehend Medical has that the entity is accurately linked to an ICD-10-CM concept.

The following truncated JSON response contains only the `modifierExtension` element.

```
"modifierExtension": [{  
  "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-score",  
  "valueDecimal": 0.94417894  
},  
{  
  "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-Concept-Score",  
  "valueDecimal": 0.8458298  
},  
{  
  "url": "http://healthlake.amazonaws.com/system-generated-linkage",  
  "valueReference": {  
    "reference": "Linkage/e366d29f-2c22-4c19-866e-09603937935a"  
  }  
},  
{  
  "url": "http://healthlake.amazonaws.com/source-document-reference",  
  "valueReference": {  
    "reference": "DocumentReference/0e938f03-da7f-4178-acd8-eea9586c46ed"  
  }  
}  
}]
```

Full JSON Response

```
{
  "subject": {
    "reference": "Patient/0679b7b7-937d-488a-b48d-6315b8e7003b"
  },
  "resourceType": "Condition",
  "code": {
    "coding": [{
      "system": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/",
      "code": "I70.0",
      "display": "Atherosclerosis of aorta"
    }],
    "text": "Atherosclerosis of aorta"
  },
  "meta": {
    "lastUpdated": "2022-10-21T19:38:30.877Z",
    "tag": [{
      "display": "SYSTEM_GENERATED"
    }]
  },
  "modifierExtension": [{
    "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-score",
    "valueDecimal": 0.94417894
  },
  {
    "url": "http://healthlake.amazonaws.com/aws-cm/infer-icd10/aws-cm-icd10-entity-Concept-Score",
    "valueDecimal": 0.8458298
  },
  {
    "url": "http://healthlake.amazonaws.com/system-generated-linkage",
    "valueReference": {
      "reference": "Linkage/e366d29f-2c22-4c19-866e-09603937935a"
    }
  },
  {
    "url": "http://healthlake.amazonaws.com/source-document-reference",
    "valueReference": {
      "reference": "DocumentReference/0e938f03-da7f-4178-acd8-eea9586c46ed"
    }
  }
],
```

```
"id": "b06d343d-ddb8-4f36-82cb-853fcd434dfd"
}
```

Example 2: A DocumentReference that contains MedicationStatement resource type

Here is an example of a clinical note based off of a patient's encounter with a medical professional.

Synthetic data

The text in this example is synthetic content and does not contain protected health information (PHI).

```
Tom is not prescribed Advil
```

The following tabs show how the ingested medical record is reported in your HealthLake data store based on the resource type.

DocumentReference

To see the results for a single DocumentReference resource type, make a GET request where the ID of a specific resource is provided.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/
r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/DocumentReference/c549125d-a218-421f-
b8bf-23614c5e796c
```

When successful, you get a 200 HTTP response code and the following truncated JSON response.

The key-value pair, "url": "http://healthlake.amazonaws.com/system-generated-resources/", indicates that the resource types inside this extension have been added by Amazon Comprehend Medical API operations. You can see the new Linkage resource type, and multiple MedicationStatement resources.

```
"extension": [{
  "extension": [{
    "url": "http://healthlake.amazonaws.com/linkage",
```

```

    "valueReference": {
      "reference": "Linkage/394bb244-177b-4409-8657-26b20ed56dd7"
    }
  },
  {
    "url": "http://healthlake.amazonaws.com/nlp-entity",
    "valueReference": {
      "reference": "MedicationStatement/cbf6af10-b0b9-451c-bdde-99611e3498a8"
    }
  },
  {
    "url": "http://healthlake.amazonaws.com/nlp-entity",
    "valueReference": {
      "reference": "MedicationStatement/9a89b0d3-6681-45ca-9926-27951edce5c7"
    }
  },
  {
    "url": "http://healthlake.amazonaws.com/nlp-entity",
    "valueReference": {
      "reference": "MedicationStatement/4a01f6c8-5f3a-4122-80ab-405312f96aa2"
    }
  },
  {
    "url": "http://healthlake.amazonaws.com/nlp-entity",
    "valueReference": {
      "reference": "MedicationStatement/fbfb77d8-70cf-4579-b4c0-d6fe3c01656b"
    }
  },
  {
    "url": "http://healthlake.amazonaws.com/nlp-entity",
    "valueReference": {
      "reference": "MedicationStatement/1340c9ce-9c48-4bf9-9b2f-d0ab027f5e0b"
    }
  }
],
"url": "http://healthlake.amazonaws.com/system-generated-resources/"
}

```

Linkage

To see the results for a single Linkage resource type, make a GET request where the ID of a specific resource is provided.


```
GET https://https://healthlake.region.amazonaws.com/datastore/datastoreId/
r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/Linkage/394bb244-177b-4409-8657-26b20ed56dd7
```

When successful, you get a 200 HTTP response code and the following JSON response.

The response contains the `item` element. In it, the key-value pair `"type": "source"` indicates the specific `DocumentReference` entry used to modify the `MedicationStatement` resource types.

You can also see the meta element and a corresponding key-value pair, `"tag": [{"display": "SYSTEM_GENERATED"}]`, indicating that these resources were created by HealthLake.

```
{
  "resourceType": "Linkage",
  "id": "394bb244-177b-4409-8657-26b20ed56dd7",
  "active": true,
  "item": [{
    "type": "alternate",
    "resource": {
      "reference": "MedicationStatement/cbf6af10-b0b9-451c-bdde-99611e3498a8",
      "type": "MedicationStatement"
    }
  },
  {
    "type": "alternate",
    "resource": {
      "reference": "MedicationStatement/9a89b0d3-6681-45ca-9926-27951edce5c7",
      "type": "MedicationStatement"
    }
  },
  {
    "type": "alternate",
    "resource": {
      "reference": "MedicationStatement/4a01f6c8-5f3a-4122-80ab-405312f96aa2",
      "type": "MedicationStatement"
    }
  },
  {
    "type": "alternate",
    "resource": {
      "reference": "MedicationStatement/fbfb77d8-70cf-4579-b4c0-d6fe3c01656b",
```

```

    "type": "MedicationStatement"
  },
  {
    "type": "alternate",
    "resource": {
      "reference": "MedicationStatement/1340c9ce-9c48-4bf9-9b2f-d0ab027f5e0b",
      "type": "MedicationStatement"
    }
  },
  {
    "type": "source",
    "resource": {
      "reference": "DocumentReference/c549125d-a218-421f-b8bf-23614c5e796c",
      "type": "DocumentReference"
    }
  }
],
"meta": {
  "lastUpdated": "2022-10-24T20:05:03.501Z",
  "tag": [{
    "display": "SYSTEM_GENERATED"
  }]
}
}

```

MedicationStatement

To see the results for a single MedicationStatement resource type, make a GET request where the ID of a specific resource is provided.

```

GET https://healthlake.region.amazonaws.com/
datastore/datastoreId/r4/eeb8005725ae22b35b4edbd68cf2dfd/r4/
MedicationStatement/9a89b0d3-6681-45ca-9926-27951edce5c7

```

The MedicationStatement resource type is where the results of the Amazon Comprehend Medical InferRxNorm API operation are found. The results are amended to the following elements: medicationCodeableConcept, meta, and modifierExtension.

medicationCodeableConcept

An element of type CodeableConcept. To learn more, see [CodeableConcept](#) in the FHIR R4 documentation.

HealthLake appends the following three key-value pairs.

- "system": "http://healthlake.amazonaws.com/aws-cm/infer-rxnorm/": Where the URL refers to a specific Amazon Comprehend Medical API operation. In this case, InferRxNorm.
- "code": "731533": Where 731533 is an RxNorm concept ID, also known as the RxCUI.
- "display": "ibuprofen 200 MG Oral Capsule [Advil]": Where ibuprofen 200 MG Oral Capsule [Advil] is the description of the RxNorm concept.

The following truncated JSON response contains only the MedicationStatement element.

```
"medicationCodeableConcept": {
  "coding": [
    {
      "system": "http://healthlake.amazonaws.com/aws-cm/infer-rxnorm/",
      "code": "731533",
      "display": "ibuprofen 200 MG Oral Capsule [Advil]"
    }
  ]
}
```

meta

The meta element contains metadata that indicates whether the code element contains details that have been added by the Amazon Comprehend Medical API operations.

The following truncated JSON response contains only the meta element.

```
"meta": {
  "lastUpdated": "2022-10-24T20:05:02.800Z",
  "tag": [
    {
      "display": "SYSTEM_GENERATED"
    }
  ]
}
```

modifierExtension

The `modifierExtension` element contains key-value pairs that provide a link back to the original `DocumentReference` used to generate the results and the related `Linkage` resource type.

```
"modifierExtension": [  
  {  
    "url": "http://healthlake.amazonaws.com/system-generated-linkage",  
    "valueReference": {  
      "reference": "Linkage/394bb244-177b-4409-8657-26b20ed56dd7"  
    }  
  },  
  {  
    "url": "http://healthlake.amazonaws.com/source-document-reference",  
    "valueReference": {  
      "reference": "DocumentReference/c549125d-a218-421f-b8bf-23614c5e796c"  
    }  
  }  
]
```

Querying HealthLake data with Amazon Athena

During a HealthLake import job, nested FHIR JSON data undergoes an ETL process and is stored in [Apache Iceberg open table format](#), where each FHIR resource type is represented as an individual table in Athena. This enables users to query the FHIR data using SQL, but without having to export it first. This is valuable, as it empowers clinicians and scientists to query FHIR data to validate their decisions or advance their research. For more information about how Apache Iceberg tables function in Athena, see [Query Apache Iceberg tables](#) in the *Athena User Guide*.

Note

HealthLake supports FHIR R4 read interaction on your HealthLake data in Athena. For more information, see [Reading a FHIR resource](#).

The topics in this section describe how to connect your HealthLake data store to Athena, how to query it using SQL, and how to connect results with other AWS services for further analysis.

Topics

- [Getting started with Amazon Athena](#)

- [Querying HealthLake data with SQL](#)
- [Example SQL queries with complex filtering](#)

Getting started with Amazon Athena

To integrate HealthLake with Amazon Athena, you must set up permissions. To do this, you'll create an Athena user, group or role, and grant them access to FHIR resources located within a HealthLake data store.

- [Granting a user, group, or role access to a HealthLake data store \(AWS Lake Formation Console\)](#)
- [Setting up an Athena account](#)

Granting a user, group, or role access to a HealthLake data store (AWS Lake Formation Console)

Persona: HealthLake administrator

The HealthLake administrator persona is a data lake administrator in AWS Lake Formation. They grant access to HealthLake data stores in Lake Formation.

For each data store created, there are two entries visible in the AWS Lake Formation console. One entry is a *resource link*. Resource link names are always displayed in *italics*. Each resource link is displayed with the name and owner of its linked shared resource. For all HealthLake data stores, the shared resource owner is the HealthLake service account. The other entry is the HealthLake data store in the HealthLake service account. The steps in this procedure use the data store that is the resource link.

To learn more about resource links, see [How resource links work in Lake Formation](#) in the *AWS Lake Formation Developer Guide*.

For a user, group, or role to be able to query data in Athena, you must grant **Describe** permission on the resource database. Then, you must grant **Select** and **Describe** on the tables.

STEP 1: To grant DESCRIBE permissions on a HealthLake data store resource link database

1. Open the AWS Lake Formation console: <https://console.aws.amazon.com/lakeformation/>
2. In the primary navigation bar, choose **Databases**.

3. On the **Databases** page, choose the radio button next to the name of the data store that is in italics.
4. Choose **Actions** (▼).
5. Choose **Grant**.
6. On the **Grant data permissions** page, under **Principals**, choose **IAM users or roles**.
7. Under **IAM users or roles**, use the **down arrow** (▼), or search for the IAM user, role, or group that you want to be able to make queries on in Athena.
8. Under **LF-Tags or catalog resources** card, choose the **Named data catalog resources** option.
9. Under **Databases**, use the **down arrow** (▼) to choose the HealthLake data store database that you want to share access to.
10. In the **Resource link permissions** card, under **Resource link permissions**, choose **Describe**.

When the grant is successful, the **Grant permission success** banner appears. To view the permission you just granted, choose **Data lake permissions**. Find the user, group, and role in the table. Under the **Permissions** column, you will see **Describe** listed.

Now you must use **Grant on target** to grant **Select** and **Describe** on all tables in the database.

STEP 2: Grant access to all tables in a HealthLake data store resource link

1. Open the AWS Lake Formation console: <https://console.aws.amazon.com/lakeformation/>
2. In the primary navigation bar, choose **Databases**.
3. On the **Databases** page, choose the radio button next to the name of the data store that is in italics.
4. Choose **Actions** (▼).
5. Choose **Grant on target**.
6. On the **Grant data permissions** page, under **Principals**, choose **IAM users or roles**.
7. Under **IAM users or roles**, use the **down arrow** (▼) or search for the IAM user, group, or role that you want to be able to make queries on in Athena.
8. Under **LF-Tags or catalog resources** card, choose the **Named data catalog resources** option.
9. Under **Databases**, use the **down arrow** (▼) to choose the HealthLake data store database that you want to grant access to.
10. Under **Tables**, choose **All tables** to share all tables with a HealthLake user.
11. In the **Table permissions** card, under **Table permissions**, choose **Describe** and **Select**.

12. Choose **Grant**.

After choosing grant, a **Grant permissions success** banner appears. The specified user can now make queries on a HealthLake data store in Athena.

Getting started with Athena

HealthLake user

The HealthLake user will use the Athena console, AWS CLI, or AWS SDKs to query a HealthLake data store shared with them by the HealthLake administrator.

To query a data store using Athena, you must do the following three things.

- Grant the IAM user or role access to the HealthLake data store via Lake Formation. To learn more, see [Granting a user, group, or role access to a HealthLake data store \(AWS Lake Formation Console\)](#).
- Create a workgroup for your HealthLake data store.
- Designate an Amazon S3 bucket to store your query results.

To get started with Athena, add the **AmazonAthenaFullAccess** and **AmazonS3FullAccess** AWS managed policies to your user, group or role. Using an AWS managed policy is great way to get started using a new service. Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they are available for use by all AWS customers. When you set permissions with IAM policies, grant only the permissions required to perform a task. To learn more about IAM and applying least-privilege, see [Apply least-privilege permissions](#) in the *IAM User Guide*.

Important

To query a HealthLake data store in Athena, you must use **Athena engine version 3**.

Workgroups are resources, and therefore you can use IAM-based policies to control access to specific workgroups. To learn more, see [Using workgroups to control query access and costs](#) in the *Athena User Guide*.

To learn more about setting up workgroups, see <https://docs.aws.amazon.com/athena/latest/ug/workgroups-procedure.html> in the *Athena User Guide*.

Note

The region your Amazon S3 bucket is in and the Athena console must match.

Before you can run a query, a query result bucket location in Amazon S3 must be specified, or you must use a workgroup that has specified a bucket and whose configuration overrides client settings. Output files are saved automatically for every query that runs.

For more details on specifying query result locations in the Athena console, see [Specifying a query result location using the Athena console](#) in the *Amazon Athena User Guide*.

To see examples of how to query your HealthLake data store in Athena, see [Querying HealthLake data with SQL](#).

Querying HealthLake data with SQL

When you import your FHIR data into HealthLake data store, the nested JSON FHIR data simultaneously undergoes an ETL process and is stored in Apache Iceberg open table format in Amazon S3. Each FHIR resource type from your HealthLake data store is converted into a table, where it can be queried using Amazon Athena. The tables can be queried individually or as group using SQL-based queries. Because of the structure of data stores, your data is imported into Athena as multiple different data types. To learn more about creating SQL queries that can access these data types, see [Query arrays with complex types and nested structures](#) in the *Amazon Athena User Guide*.

Note

All examples in this topic use fictionalized data created using Synthea. To learn more about creating a data store preloaded with Synthea data, see [Creating a HealthLake data store](#).

For each element in a resource type, the FHIR specification defines a cardinality. The cardinality of an element defines the lower and upper bounds of how many times this element can appear. When constructing a SQL query, you must take this into account. For example, let's look at some elements in [Resource type: Patient](#).

- **Element: Name** The FHIR specification sets the cardinality as 0..*.

The element is captured as an array.

```
[{
  id = null,
  extension = null,
  use = official,
  _use = null,
  text = null,
  _text = null,
  family = Wolf938,
  _family = null,
  given = [Noel608],
  _given = null,
  prefix = null,
  _prefix = null,
  suffix = null,
  _suffix = null,
  period = null
}]
```

In Athena, to see how a resource type has been ingested, search for it under **Tables and views**. To access elements in this array, you can use dot notation. Here's a simple example that would access the values for given and family.

```
SELECT
    name[1].given as FirstName,
    name[1].family as LastName
FROM Patient
```

- **Element: MaritalStatus** The FHIR specification sets the cardinality as 0..1.

This element is captured as JSON.

```
{
  id = null,
  extension = null,
  coding = [
    {
      id = null,
      extension = null,
```

```
    system = http: //terminology.hl7.org/CodeSystem/v3-MaritalStatus,  
    _system = null,  
    version = null,  
    _version = null,  
    code = S,  
    _code = null,  
    display = Never Married,  
    _display = null,  
    userSelected = null,  
    _userSelected = null  
  }  
  
],  
  text = Never Married,  
  _text = null  
}
```

In Athena, to see how a resource type has been ingested, search for it under **Tables and views**. To access key-value pairs in the JSON, you can use dot notation. Because it isn't an array, no array index is required. Here's a simple example that would access the value for text.

```
SELECT  
    maritalstatus.text as MaritalStatus  
FROM Patient
```

To learn more about accessing and searching JSON, see [Querying JSON](#) in the *Athena User Guide*.

Athena Data Manipulation Language (DML) query statements are based on Trino. Athena does not support all of Trino's features, and there are *significant* differences. To learn more, see [DML queries, functions, and operators](#) in the *Amazon Athena User Guide*.

Furthermore, Athena supports multiple data types that you may encounter when creating queries of your HealthLake data store. To learn more about data types in Athena, see [Data types in Amazon Athena](#) in the *Amazon Athena User Guide*.

To learn more about how SQL queries work in Athena, see [SQL reference for Amazon Athena](#) in the *Amazon Athena User Guide*.

Each tab shows examples of how to search on the specified resource types and associated elements using Athena.

Element: Extension

The element extension is used to create custom fields in a data store.

This example shows you how to access the features of the extension element found in the Patient resource type.

When your HealthLake data store is imported into Athena, the elements of a resource type are parsed differently. Because the structure of the element is variable, it cannot be fully specified in the schema. To handle that variability, the elements inside the array are passed as strings.

In the table description of Patient, you can see the element extension described as `array<string>`, which means you can access the elements of array by using an index value. To access the elements of the string, however, you must use `json_extract`.

Here is a single entry from the extension element found in the patient table.

```
[{
  "valueString": "Kerry175 Cumerata161",
  "url": "http://hl7.org/fhir/StructureDefinition/patient-mothersMaidenName"
},
{
  "valueAddress": {
    "country": "DE",
    "city": "Hamburg",
    "state": "Hamburg"
  },
  "url": "http://hl7.org/fhir/StructureDefinition/patient-birthPlace"
},
{
  "valueDecimal": 0.0,
  "url": "http://synthetichealth.github.io/synthea/disability-adjusted-life-years"
},
{
  "valueDecimal": 5.0,
  "url": "http://synthetichealth.github.io/synthea/quality-adjusted-life-years"
}
]
```

Even though this is valid JSON, Athena treats it as a string.

This SQL query example demonstrates how you can create a table that contains the patient-mothersMaidenName and patient-birthPlace elements. To access these elements, you need to use different array indices and `json_extract`.

```
SELECT
    extension[1],
    json_extract(extension[1], '$.valueString') AS MothersMaidenName,
    extension[2],
    json_extract(extension[2], '$.valueAddress.city') AS birthPlace
FROM patient
```

To learn more about queries that involve JSON, see [Extracting data from JSON](#) in the *Amazon Athena User Guide*.

Element: birthDate (Age)

Age is *not* an element of the Patient resource type in FHIR. Here are two examples for searches that filter based on age.

Because age is not an element, we use the birthDate for the SQL queries. To see how an element has been ingested into FHIR, search for the table name under **Tables and views**. You can see that it is of type **string**.

Example 1: Calculating a value for age

In this sample SQL query, we use a built-in SQL tool, `current_date` and `year` to extract those components. Then, we subtract them to return a patient's actual age as a column called age.

```
SELECT
    (year(current_date) - year(date(birthdate))) as age
FROM patient
```

Example 2: Filtering for patients who are born before 2019-01-01 and are male.

The SQL query shows you how to use the `CAST` function to cast the birthDate element as type `DATE`, and how to filter based on two criteria in the `WHERE` clause. Because the element is ingested as type **string** by default, we must `CAST` it as type `DATE`. Then you can use the `<` operator to compare it to a different date, 2019-01-01. By using `AND`, you can add a second criteria to the `WHERE` clause.

```
SELECT birthdate
```

```
FROM patient
-- we convert birthdate (varchar) to date > cast that as date too
WHERE CAST(birthdate AS DATE) < CAST('2019-01-01' AS DATE) AND gender = 'male'
```

Resource type: Location

This example shows searches for locations within the Location resource type where the city name is Attleboro.

```
SELECT *
FROM Location
WHERE address.city='ATTLEBORO'
LIMIT 10;
```

Element: Age

```
SELECT birthdate
FROM patient
-- we convert birthdate (varchar) to date > cast that as date too
WHERE CAST(birthdate AS DATE) < CAST('2019-01-01' AS DATE) AND gender = 'male'
```

Resource type: Condition

The resource type condition stores diagnosis data related to issues that have risen to a level of concern. HealthLake's integrated medical natural language processing (NLP) generates *new* Condition resources based on details found in the DocumentReference resource type. When new resource are generated, HealthLake appends the tag SYSTEM_GENERATED to the meta element. This sample SQL query demonstrates how you can search the condition table and return results where the SYSTEM_GENERATED results have been removed.

To learn more about HealthLake's integrated natural language processing (NLP), see [Integrated natural language processing \(NLP\) for HealthLake](#).

```
SELECT *
FROM condition
WHERE meta.tag[1] is NULL
```

You can also search within a specified string element to filter your query further. The `modifierextension` element contains details about which DocumentReference resource

was used to generate a set of conditions. Again, you must use `json_extract` to access the nested JSON elements that are brought into Athena as a string.

This sample SQL query demonstrates how you can search for all the `Condition` that has been generated based off of a specific `DocumentReference`. Use `CAST` to set the JSON element as a string so that you can use `LIKE` to compare.

```
SELECT
    meta.tag[1].display as SystemGenerated,
    json_extract(modifierextension[4], '$.valueReference.reference') as
    DocumentReference
FROM condition
WHERE meta.tag[1].display = 'SYSTEM_GENERATED'

AND CAST(json_extract(modifierextension[4], '$.valueReference.reference') as
    VARCHAR) LIKE '%DocumentReference/67aa0278-8111-40d0-8adc-43055eb9d18d%'
```

Resource type: Observation

The resource type, `Observation` stores measurements and simple assertions made about a patient, device, or other subject. HealthLake's integrated natural language processing (NLP) generates *new* `Observation` resources based on details found in a `DocumentReference` resource. This sample SQL query includes `WHERE meta.tag[1] is NULL` commented out, which means that the `SYSTEM_GENERATED` results are included.

```
SELECT valueCodeableConcept.coding[1].code
FROM Observation
WHERE valueCodeableConcept.coding[1].code = '266919005'
-- WHERE meta.tag[1] is NULL
```

This column was imported as an [struct](#). Therefore, you can access elements inside it using dot notation.

Resource type: MedicationStatement

`MedicationStatement` is a FHIR resource type that you can use to store details about medications a patient has taken, is taking, or will take in the future. HealthLake's integrated medical natural language processing (NLP) generates new `MedicationStatement` resources based on documents found in the `DocumentReference` resource type. When new resources are generated, HealthLake appends the tag `SYSTEM_GENERATED` to the `meta` element. This sample

SQL query demonstrates how to create a query that filters based off of a single patient by using their identifier and finds resources that have been added by HealthLake's integrated NLP.

```
SELECT *
FROM medicationstatement
WHERE meta.tag[1].display = 'SYSTEM_GENERATED' AND subject.reference =
  'Patient/0679b7b7-937d-488a-b48d-6315b8e7003b';
```

To learn more about HealthLake's integrated natural language processing (NLP), see [Integrated natural language processing \(NLP\) for HealthLake](#).

Example SQL queries with complex filtering

The following examples demonstrate how to use Amazon Athena SQL queries with complex filtering to locate FHIR data from a HealthLake data store.

Example Create filtering criteria based on demographic data

Identifying the correct patient demographics is important when creating a patient cohort. This sample query demonstrates how you can use Trino dot notation and `json_extract` to filter data in your HealthLake data store.

```
SELECT
  id
  , CONCAT(name[1].family, ' ', name[1].given[1]) as name
  , (year(current_date) - year(date(birthdate))) as age
  , gender as gender
  , json_extract(extension[1], '$.valueString') as MothersMaidenName
  , json_extract(extension[2], '$.valueAddress.city') as birthPlace
  , maritalstatus.coding[1].display as maritalstatus
  , address[1].line[1] as addressline
  , address[1].city as city
  , address[1].district as district
  , address[1].state as state
  , address[1].postalcode as postalcode
  , address[1].country as country
  , json_extract(address[1].extension[1], '$.extension[0].valueDecimal') as latitude
  , json_extract(address[1].extension[1], '$.extension[1].valueDecimal') as longitude
  , telecom[1].value as telNumber
  , deceasedboolean as deceasedIndicator
  , deceaseddatetime
```

```
FROM database.patient;
```

Using the Athena Console, you can further sort and download the results.

Example Create filters for a patient and their related conditions

The following example query demonstrates how you can find and sort all the related conditions for the patients found in a HealthLake data store.

```
SELECT
  patient.id as patientId
    , condition.id as conditionId
    , CONCAT(name[1].family, ' ', name[1].given[1]) as name
    , condition.meta.tag[1].display
    , json_extract(condition.modifierextension[1], '$.valueDecimal') AS confidenceScore
    , category[1].coding[1].code as categoryCode
    , category[1].coding[1].display as categoryDescription
    , code.coding[1].code as diagnosisCode
    , code.coding[1].display as diagnosisDescription
    , onsetdatetime
    , severity.coding[1].code as severityCode
    , severity.coding[1].display as severityDescription
    , verificationstatus.coding[1].display as verificationStatus
    , clinicalstatus.coding[1].display as clinicalStatus
    , encounter.reference as encounterId
    , encounter.type as encountertype
FROM database.patient, condition
WHERE CONCAT('Patient/', patient.id) = condition.subject.reference
ORDER BY name;
```

You can use the Athena console to further sort the results or download them for further analysis.

Example Create filters for patients and their related observations

The following example query demonstrates how to find and sort all related observations for patients found in a HealthLake data store.

```
SELECT
  patient.id as patientId
    , observation.id as observationId
    , CONCAT(name[1].family, ' ', name[1].given[1]) as name
    , meta.tag[1].display
    , json_extract(modifierextension[1], '$.valueDecimal') AS confidenceScore
```



```

, status
, category[1].coding[1].code as categoryCode
, category[1].coding[1].display as categoryDescription
, code.coding[1].code as observationCode
, code.coding[1].display as observationDescription
, effectivedatetime
, CASE
  WHEN valuequantity.value IS NOT NULL THEN CONCAT(CAST(valuequantity.value AS
VARCHAR),' ',valuequantity.unit)
    WHEN valueCodeableConcept.coding [ 1 ].code IS NOT NULL THEN
CAST(valueCodeableConcept.coding [ 1 ].code AS VARCHAR)
    WHEN valuestring IS NOT NULL THEN CAST(valuestring AS VARCHAR)
    WHEN valueboolean IS NOT NULL THEN CAST(valueboolean AS VARCHAR)
    WHEN valueinteger IS NOT NULL THEN CAST(valueinteger AS VARCHAR)
    WHEN valueratio IS NOT NULL THEN CONCAT(CAST(valueratio.numerator.value AS
VARCHAR),'/',CAST(valueratio.denominator.value AS VARCHAR))
    WHEN valuerange IS NOT NULL THEN CONCAT(CAST(valuerange.low.value AS
VARCHAR),'-',CAST(valuerange.high.value AS VARCHAR))
    WHEN valueSampledData IS NOT NULL THEN CAST(valueSampledData.data AS VARCHAR)
    WHEN valueTime IS NOT NULL THEN CAST(valueTime AS VARCHAR)
    WHEN valueDateTime IS NOT NULL THEN CAST(valueDateTime AS VARCHAR)
    WHEN valuePeriod IS NOT NULL THEN valuePeriod.start
    WHEN component[1] IS NOT NULL THEN CONCAT(CAST(component[2].valuequantity.value
AS VARCHAR),' ',CAST(component[2].valuequantity.unit AS VARCHAR),
'/', CAST(component[1].valuequantity.value AS VARCHAR), '
',CAST(component[1].valuequantity.unit AS VARCHAR))
  END AS observationvalue
, encounter.reference as encounterId
, encounter.type as encountertype
FROM database.patient, observation
WHERE CONCAT('Patient/', patient.id) = observation.subject.reference
ORDER BY name;

```

Example Create filtering conditions for a patient and their related procedures

Connecting procedures to patients is an important aspect of healthcare. The following SQL example query demonstrates how to use FHIR Patient and Procedure resource types to accomplish this. The following SQL query will return all patients and their related procedures found in your HealthLake data store.

```

SELECT
  patient.id as patientId
, PROCEDURE.id as procedureId

```

```
, CONCAT(name[1].family, ' ', name[1].given[1]) as name
, status
, category.coding[1].code as categoryCode
, category.coding[1].display as categoryDescription
, code.coding[1].code as procedureCode
, code.coding[1].display as procedureDescription
, performeddatetime
, performer[1]
, encounter.reference as encounterId
, encounter.type as encountertype
FROM database.patient, procedure
WHERE CONCAT('Patient/', patient.id) = procedure.subject.reference
ORDER BY name;
```

You can use the Athena console to download the results for further analysis or sort them to better understand the results.

Example Create filtering conditions for a patient and their related prescriptions

Seeing a current list of medications that patients are taking is important. Using Athena, you can write a SQL query that uses both the Patient and MedicationRequest resource types found in your HealthLake data store.

The following SQL query joins the Patient and MedicationRequest tables imported into Athena. It also organizes the prescriptions into their individual entries by using dot notation.

```
SELECT
  patient.id as patientId
  , medicationrequest.id as medicationrequestid
  , CONCAT(name[1].family, ' ', name[1].given[1]) as name
  , status
  , statusreason.coding[1].code as categoryCode
  , statusreason.coding[1].display as categoryDescription
  , category[1].coding[1].code as categoryCode
  , category[1].coding[1].display as categoryDescription
  , priority
  , donotperform
  , encounter.reference as encounterId
  , encounter.type as encountertype
  , medicationcodeableconcept.coding[1].code as medicationCode
  , medicationcodeableconcept.coding[1].display as medicationDescription
  , dosageinstruction[1].text as dosage
FROM database.patient, medicationrequest
```

```
WHERE CONCAT('Patient/', patient.id ) = medicationrequest.subject.reference
ORDER BY name
```

You can use the Athena console to sort the results or download them for further analysis.

Example See medications found in the MedicationStatement resource type

The following example query shows you how to organize the nested JSON imported into Athena using SQL. The query uses the FHIR meta element to indicate when a medication has been added by HealthLake's integrated natural language processing (NLP). It also uses `json_extract` to search for data inside the array of JSON strings. For more information, see [Natural language processing](#).

```
SELECT
  medicationcodeableconcept.coding[1].code as medicationCode
  , medicationcodeableconcept.coding[1].display as medicationDescription
  , meta.tag[1].display
  , json_extract(modifierextension[1], '$.valueDecimal') AS confidenceScore
FROM medicationstatement;
```

You can use the Athena console to download these results or sort them.

Example Filter for a specific disease type

The example shows how you can find a group of patients, aged 18 to 75, who have been diagnosed with diabetes.

```
SELECT patient.id as patientId,
  condition.id as conditionId,
  CONCAT(name [ 1 ].family, ' ', name [ 1 ].given [ 1 ]) as name,
  (year(current_date) - year(date(birthdate))) AS age,
  CASE
    WHEN condition.encounter.reference IS NOT NULL THEN condition.encounter.reference
    WHEN observation.encounter.reference IS NOT NULL THEN observation.encounter.reference
  END as encounterId,
  CASE
    WHEN condition.encounter.type IS NOT NULL THEN condition.encounter.type
    WHEN observation.encounter.type IS NOT NULL THEN observation.encounter.type
  END AS encountertype,
  condition.code.coding [ 1 ].code as diagnosisCode,
  condition.code.coding [ 1 ].display as diagnosisDescription,
  observation.category [ 1 ].coding [ 1 ].code as categoryCode,
  observation.category [ 1 ].coding [ 1 ].display as categoryDescription,
```

```

observation.code.coding [ 1 ].code as observationCode,
observation.code.coding [ 1 ].display as observationDescription,
effectivedatetime AS observationDateTime,
CASE
    WHEN valuequantity.value IS NOT NULL THEN CONCAT(CAST(valuequantity.value AS
VARCHAR),' ',valuequantity.unit)
    WHEN valueCodeableConcept.coding [ 1 ].code IS NOT NULL THEN
CAST(valueCodeableConcept.coding [ 1 ].code AS VARCHAR)
    WHEN valuestring IS NOT NULL THEN CAST(valuestring AS VARCHAR)
    WHEN valueboolean IS NOT NULL THEN CAST(valueboolean AS VARCHAR)
    WHEN valueinteger IS NOT NULL THEN CAST(valueinteger AS VARCHAR)
    WHEN valueratio IS NOT NULL THEN CONCAT(CAST(valueratio.numerator.value AS
VARCHAR),'/',CAST(valueratio.denominator.value AS VARCHAR))
    WHEN valuerange IS NOT NULL THEN CONCAT(CAST(valuerange.low.value AS
VARCHAR),'-',CAST(valuerange.high.value AS VARCHAR))
    WHEN valueSampledData IS NOT NULL THEN CAST(valueSampledData.data AS VARCHAR)
    WHEN valueTime IS NOT NULL THEN CAST(valueTime AS VARCHAR)
    WHEN valueDateTime IS NOT NULL THEN CAST(valueDateTime AS VARCHAR)
    WHEN valuePeriod IS NOT NULL THEN valuePeriod.start
    WHEN component[1] IS NOT NULL THEN CONCAT(CAST(component[2].valuequantity.value
AS VARCHAR),' ',CAST(component[2].valuequantity.unit AS VARCHAR),
'/', CAST(component[1].valuequantity.value AS VARCHAR),'
',CAST(component[1].valuequantity.unit AS VARCHAR))
    END AS observationvalue,
CASE
    WHEN condition.meta.tag [ 1 ].display = 'SYSTEM GENERATED' THEN 'YES'
    WHEN condition.meta.tag [ 1 ].display IS NULL THEN 'NO'
    WHEN observation.meta.tag [ 1 ].display = 'SYSTEM GENERATED' THEN 'YES'
    WHEN observation.meta.tag [ 1 ].display IS NULL THEN 'NO'
    END AS IsSystemGenerated,
CAST(
    json_extract(
        condition.modifierextension [ 1 ],
        '$.valueDecimal'
    ) AS int
) AS confidenceScore
FROM database.patient,
database.condition,
database.observation
WHERE CONCAT('Patient/', patient.id) = condition.subject.reference
AND CONCAT('Patient/', patient.id) = observation.subject.reference
AND (year(current_date) - year(date(birthdate))) >= 18
AND (year(current_date) - year(date(birthdate))) <= 75

```

```
AND condition.code.coding [ 1 ].display like ('%diabetes%');
```

Now you can use the Athena console to sort the results or download them for further analysis.

Monitoring AWS HealthLake

Monitoring and logging are important parts of maintaining the security, reliability, availability, and performance of AWS HealthLake. AWS provides the following services to watch HealthLake, report when something is wrong, and take automatic actions when appropriate.

- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the [AWS CloudTrail User Guide](#).
- *Amazon CloudWatch* monitors your AWS resources and the applications you run on AWS in real time. You can collect and track metrics, create customized dashboards, and set alarms that notify you or take actions when a specified metric reaches a threshold that you specify. For example, you can have CloudWatch track CPU usage or other metrics of your Amazon EC2 instances and automatically launch new instances when needed. For more information, see the [Amazon CloudWatch User Guide](#).
- *Amazon EventBridge* is a serverless event bus service that makes it easy to connect your applications with data from a variety of sources. EventBridge delivers a stream of real-time data from your own applications, Software-as-a-Service (SaaS) applications, and AWS services and routes that data to targets such as Lambda. This enables you to monitor events that happen in services, and build event-driven architectures. For more information, see the [Amazon EventBridge User Guide](#).

Topics

- [Logging HealthLake API calls using AWS CloudTrail](#)
- [Monitoring HealthLake metrics using Amazon CloudWatch](#)
- [Monitoring HealthLake events using Amazon EventBridge](#)

Logging HealthLake API calls using AWS CloudTrail

AWS HealthLake is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in HealthLake. CloudTrail captures all API calls for HealthLake as events. The calls captured include calls from the HealthLake console and code calls to the HealthLake API operations. If you create a trail, you can enable continuous delivery of CloudTrail

events to an Amazon S3 bucket, including events for HealthLake. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to HealthLake, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the [AWS CloudTrail User Guide](#).

AWS HealthLake Information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in HealthLake, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see [Viewing Events with CloudTrail Event History](#).

For an ongoing record of events in your AWS account, including events for HealthLake, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- [Overview for Creating a Trail](#)
- [CloudTrail Supported Services and Integrations](#)
- [Configuring Amazon SNS Notifications for CloudTrail](#)
- [Receiving CloudTrail Log Files from Multiple Regions](#) and [Receiving CloudTrail Log Files from Multiple Accounts](#)

All HealthLake actions are logged by CloudTrail and are documented in the [HealthLake API Reference](#) and in this Developer Guide for actions performed using the FHIR REST API. For example, calls to the following actions generate entries in the CloudTrail log files:

- `DescribeFHIRImportJob`
- `DescribeFHIRExportJob`
- `StartFHIRImportJob`
- `ListFHIRImportJobs`
- `StartFHIRExportJob`

- ListFHIRExportJobs
- CreateFHIRDatastore
- ListFHIRDatastores
- DeleteFHIRDatastore
- DescribeFHIRDatastore
- UpdateResource
- CreateResource
- DeleteResource
- ReadResource
- GetCapabilities
- SearchWithGet
- SearchWithPost

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the [CloudTrail userIdentity Element](#).

Understanding AWS HealthLake Log File Entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the CreateFHIRDatastore action.

```
{
```



```

    "eventVersion": "1.08",
    "userIdentity": {
      "type": "AssumedRole",
      "principalId": "AR0A2B3ZH0ADD20J4AHJX:git
full_access_iam_role580074395690222150",
      "arn": "arn:aws:sts::691207106566:assumed-role/
colossusfrontend_full_access_iam_role/_iam_role580074395690222150",
      "accountId": "AccountID",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "sessionContext": {
        "sessionIssuer": {
          "type": "Role",
          "principalId": "AR0A2B3ZH0ADD20J4AHJX",
          "arn": "arn:aws:iam::691207106566:role/full_access_iam_role",
          "accountId": "AccountID",
          "userName": "full_access_iam_role"
        },
        "webIdFederationData": {

        },
        "attributes": {
          "mfaAuthenticated": "false",
          "creationDate": "2020-11-20T00:08:15Z"
        }
      }
    },
    "eventTime": "2020-11-20T00:08:16Z",
    "eventSource": "healthlake.amazonaws.com",
    "eventName": "CreateFHIRDatastore",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "3.213.247.1",
    "userAgent": "Coral/Netty4",
    "requestParameters": {
      "datastoreName":
"testCreateFHIRDatastore_GBYAZFCLLBLSUT0YYFQZRLBLQJNFOYQVRPZB0JAIIUAHICAEAGIWLNVQEYAMSXVWMBLXC",
      "datastoreTypeVersion": "R4",
      "clientToken": "d737ffe0-14dd-44cc-9f0a-fdf59b26c66b"
    },
    "responseElements": {
      "datastoreId": "datastoreID",
      "datastoreArn": "arn:aws:healthlake:us-
east-1:691207106566:datastore/55576c487ff4975262b10d1d65eb4509",
      "datastoreStatus": "CREATING",
      "datastoreEndpoint": "datastore_endpoint/"
    }
  }
}

```

```
},
"requestID": "68e62bdd-d2d4-44c1-af69-e6f055a69f99",
"eventID": "7ef483dc-5dca-469e-823a-7d9e3a7fe924",
"readOnly": false,
"eventType": "AwsApiCall",
"managementEvent": true,
"eventCategory": "Management",
"recipientAccountId": "691207106566"
}
```

Monitoring HealthLake metrics using Amazon CloudWatch

You can monitor HealthLake using Amazon CloudWatch, which collects raw data and processes it into readable, near real-time metrics. These statistics are kept for 15 months, so you can use that historical information and gain a better perspective on how your web application or service is performing. You can also set alarms that watch for certain thresholds, and send notifications or take actions when those thresholds are met. For more information, see the [Amazon CloudWatch User Guide](#).

Note

Metrics are reported for all [native HealthLake actions](#).

The following tables list HealthLake metrics and dimensions reported to CloudWatch. Each is presented as a frequency count for a user-specified data range.

The following HealthLake metrics are reported to CloudWatch.

HealthLake metrics reported to CloudWatch

Metric	Description
Call Count	The number of calls to APIs. This can be reported for the account or a specified data store. Units: Count Valid Statistics: Sum, Count

Metric	Description
	Dimensions: Operation, datastore ID, data store type
Successful Requests	The number of successful API requests. Units: Count Valid Statistics: Sum, Average Dimensions: Operation, data store ID, data store type
User Errors	The number of requests that failed due to user error. Units: Count Valid Statistics: Sum, Average Dimensions: Operation, data store ID, data store type
Server Errors	The number of requests that failed due to server error. Units: Count Valid Statistics: Sum, Average Dimensions: Operation, data store ID, data store type

Metric	Description
Throttled Requests	The number of requests that have been throttled. This metric is not included in user or server errors counts. Units: Count Valid Statistics: Sum, Average Dimensions: Operation, data store ID, data store type
Latency	The time it took in milliseconds to process the user request. Unit: Milliseconds Valid statistics: Minimum, Maximum, Average Dimensions: Operation, data store ID, data store type

The following HealthLake dimensions are reported to CloudWatch.

HealthLake Dimensions reported to CloudWatch

Dimension	Description
Operation	The API operation used in the request
DataStoreID	The data store ID used in the request
DataStoreType	The type of data store used in the request

You can get metrics for HealthLake with the AWS Management Console, the AWS CLI, or the CloudWatch API. You can use the CloudWatch API through one of the Amazon AWS Software Development Kits (SDKs) or the CloudWatch API tools. The HealthLake console displays graphs based on the raw data from the CloudWatch API.

You must have the appropriate CloudWatch permissions to monitor HealthLake with CloudWatch. For more information, see [Identity and access management for Amazon CloudWatch](#) in the *Amazon CloudWatch User Guide*.

Viewing HealthLake metrics

To view metrics (CloudWatch console)

1. Sign in to the AWS Management Console and open the [CloudWatch console](#).
2. Choose **Metrics**, choose **All Metrics**, and then choose **AWS/HealthLake**.
3. Choose the dimension, choose a metric name, then choose **Add to graph**.
4. Choose a value for the date range. The metric count for the selected date range is displayed in the graph.

Creating an alarm using CloudWatch

A CloudWatch alarm watches a single metric over a specified time period, and performs one or more actions: sending a notification to an Amazon Simple Notification Service (SNS) topic or Auto Scaling policy. The action or actions are based on the value of the metric relative to a given threshold over a number of time periods that you specify. CloudWatch can also send you an SNS message when the alarm changes state.

Note

CloudWatch alarms invoke actions only when the state changes and has persisted for the period you specify.

To view metrics (CloudWatch console)

1. Sign in to the [CloudWatch console](#).
2. Choose **Alarms**, and then choose **Create Alarm**.
3. Choose **AWS/HealthLake**, and then choose a metric.
4. For **Time Range**, choose a time range to monitor, and then choose **Next**.
5. Enter a **Name** and **Description**.
6. For **Whenever**, choose **>=**, and type a maximum value.

7. If you want CloudWatch to send an email when the alarm state is reached, in the Actions section, for Whenever this alarm, choose State is **ALARM**. For Send notification to, choose a mailing list or choose **New list** and create a new mailing list.
8. Preview the alarm in the Alarm Preview section. If you are satisfied with the alarm, choose **Create Alarm**.

Monitoring HealthLake events using Amazon EventBridge

Amazon EventBridge is a serverless service that uses events to connect application components together, making it easier for you to build scalable event-driven applications. The basis of EventBridge is to create [rules](#) that route [events](#) to [targets](#). AWS HealthLake provides durable delivery of state changes to EventBridge. For more information, see [What is Amazon EventBridge?](#) in the *Amazon EventBridge User Guide*.

Note

To learn how to send HealthLake events to Amazon EventBridge, see [Amazon EventBridge integration for AWS HealthLake](#) in the *AWS for Industries* blog.

Topics

- [HealthLake events sent to EventBridge](#)
- [HealthLake event structure](#)

HealthLake events sent to EventBridge

The following table lists all HealthLake events sent to EventBridge for processing.

HealthLake event type	State
Data store events	
Data Store Creating	CREATING
Data Store Active	ACTIVE
Data Store Deleting	DELETING

HealthLake event type	State
Data Store Deleted	DELETED
For more information, see datastoreStatus in the <i>AWS HealthLake API Reference</i> .	

Import job events

Import Job Submitted	SUBMITTED
Import Job In Progress	IN_PROGRESS
Import Job Completed With Errors	COMPLETED_WITH_ERRORS
Import Job Completed	COMPLETED
Import Job Failed	FAILED

For more information, see [jobStatus](#) in the *AWS HealthLake API Reference*.

Export job events

Export Job Submitted	SUBMITTED
Export Job In Progress	IN_PROGRESS
Export Job Completed With Errors	COMPLETED_WITH_ERRORS
Export Job Completed	COMPLETED
Export Job Failed	FAILED

For more information, see [jobStatus](#) in the *AWS HealthLake API Reference*.

HealthLake event structure

HealthLake events are objects with JSON structure that also contain metadata details. You can use the metadata as input to either recreate an event or learn more information. All associated metadata fields are listed in a table under the code examples in the following menus. For more information, see [AWS service event metadata](#) in the *Amazon EventBridge User Guide*.

Note

To learn how to send HealthLake events to Amazon EventBridge, see [Amazon EventBridge integration for AWS HealthLake](#) in the *AWS for Industries* blog.

Data store events**Data Store Creating****State - CREATING**

```
{
  "version": "0",
  "id": "514ad836-bb8a-4523-a10b-fa2756c3bdb0",
  "detail-type": "Data Store Creating",
  "source": "aws.healthlake",
  "account": "123456789012",
  "time": "2023-12-08T08:58:12Z",
  "region": "us-east-1",
  "resources":
  [
    "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/eeb8005725ae22b35b4edbdc68cf2dfd"
  ],
  "detail":
  {
    "datastoreId": "eeb8005725ae22b35b4edbdc68cf2dfd",
    "datastoreName": "your-data-store-name",
    "datastoreTypeVersion": "R4",
    "datastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/eeb8005725ae22b35b4edbdc68cf2dfd/r4/"
  }
}
```

Data Store Active**State - ACTIVE**

```
{
  "version": "0",
  "id": "d57105bc-0d2d-4009-b34d-453e2567c599",
```



```

    "detail-type": "Data Store Active",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T09:16:51Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd68cf2dfd"
    ],
    "detail":
    {
        "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
        "datastoreName": "your-data-store-name",
        "datastoreTypeVersion": "R4",
        "datastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
eeb8005725ae22b35b4edbd68cf2dfd/r4/"
    }
}

```

Data Store Deleting

State - DELETING

```

{
    "version": "0",
    "id": "d135ee1f-e14a-4730-8766-7b98f822c94a",
    "detail-type": "Data Store Deleting",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T12:44:47Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd68cf2dfd"
    ],
    "detail":
    {
        "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
        "datastoreName": "your-data-store-name",
        "datastoreTypeVersion": "R4",
        "datastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/
eeb8005725ae22b35b4edbd68cf2dfd/r4/"
    }
}

```

```
}  
}
```

Data Store Deleted

State - DELETED

```
{  
  "version": "0",  
  "id": "6d880b86-e115-4947-81a9-494db704571a",  
  "detail-type": "Data Store Deleted",  
  "source": "aws.healthlake",  
  "account": "123456789012",  
  "time": "2023-05-12T12:58:03Z",  
  "region": "us-east-1",  
  "resources":  
  [  
    "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/  
eeb8005725ae22b35b4edbd6c68cf2dfd"  
  ],  
  "detail":  
  {  
    "datastoreId": "eeb8005725ae22b35b4edbd6c68cf2dfd",  
    "datastoreName": "your-data-store-name",  
    "datastoreTypeVersion": "R4",  
    "datastoreEndpoint": "https://healthlake.us-east-1.amazonaws.com/datastore/  
eeb8005725ae22b35b4edbd6c68cf2dfd/r4/"  
  }  
}
```

Data store events - metadata descriptions

Name	Type	Description
version	string	The EventBridge event schema version.
id	string	The Version 4 UUID generated for every event.

Name	Type	Description
detail-type	string	The type of event that is being sent.
source	string	Identifies the service that generated the event.
account	string	The 12-digit AWS account ID of the data store owner.
time	string	The time the event occurred.
region	string	Identifies the AWS Region of the data store.
resources	array (string)	A JSON array that contains the ARN of the data store.
detail	object	A JSON object that contains information about the event.
detail.datastoreId	string	The data store ID associated with the status change event.
detail.datastoreName	string	The data store name.
detail.datastoreTypeVersion	string	The data store FHIR version.
detail.datastoreEndpoint	string	The data store endpoint.

Import job events

Import Job Submitted

State - SUBMITTED

```
{
```

```

    "version": "0",
    "id": "25e606f7-800c-da41-45df-0e68587250c9",
    "detail-type": "Import Job Submitted",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T01:50:51Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd68cf2dfd"
    ],
    "detail":
    {
        "jobId": "08c60716d6321710893ff88410e902c2",
        "submitTime": "2023-12-08T01:50:50.986Z",
        "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
        "inputDataConfig":
        {
            "s3Uri": "s3://amzn-s3-demo-source-bucket/input/"
        }
    }
}

```

Import Job In Progress

State - IN_PROGRESS

```

{
    "version": "0",
    "id": "cc886b49-2737-19c4-7c4e-84ac9429ab73",
    "detail-type": "Import Job In Progress",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T01:51:23Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd68cf2dfd"
    ],
    "detail":
    {
        "jobId": "08c60716d6321710893ff88410e902c2",

```

```

    "submitTime": "2023-12-08T01:50:50.986Z",
    "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
    "inputDataConfig":
    {
        "s3Uri": "s3://amzn-s3-demo-source-bucket/input/"
    }
}

```

Import Job Completed

State - COMPLETED

```

{
    "version": "0",
    "id": "36c865ef-da41-76ef-c882-3ba8dad8656b",
    "detail-type": "Import Job Completed",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T02:14:42Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/eeb8005725ae22b35b4edbd68cf2dfd"
    ],
    "detail":
    {
        "jobId": "08c60716d6321710893ff88410e902c2",
        "submitTime": "2023-12-08T01:50:50.986Z",
        "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
        "inputDataConfig":
        {
            "s3Uri": "s3://amzn-s3-demo-source-bucket/input/"
        }
    }
}

```

Import Job Completed With Errors

State - COMPLETED_WITH_ERRORS

```

{

```

```

    "version": "0",
    "id": "b61387d7-bffe-4f01-8291-65dc4be52cc1",
    "detail-type": "Import Job Completed With Errors",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T02:14:42Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd68cf2dfd"
    ],
    "detail":
    {
        "jobId": "08c60716d6321710893ff88410e902c2",
        "submitTime": "2023-12-08T01:50:50.986Z",
        "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
        "inputDataConfig":
        {
            "s3Uri": "s3://amzn-s3-demo-source-bucket/input/"
        }
    }
}

```

Import Job Failed

State - FAILED

```

{
    "version": "0",
    "id": "c4d65575-d1a7-4040-9c6c-c225bf6723c5",
    "detail-type": "Import Job Failed",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T02:14:42Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd68cf2dfd"
    ],
    "detail":
    {
        "jobId": "08c60716d6321710893ff88410e902c2",

```

```
    "submitTime": "2023-12-08T01:50:50.986Z",
    "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
    "inputDataConfig":
    {
        "s3Uri": "s3://amzn-s3-demo-source-bucket/input/"
    }
}
```

Import job events - metadata descriptions

Name	Type	Description
version	string	The EventBridge event schema version.
id	string	The Version 4 UUID generated for every event.
detail-type	string	The type of event that is being sent.
source	string	Identifies the service that generated the event.
account	string	The 12-digit AWS account ID of the data store owner.
time	string	The time the event occurred.
region	string	Identifies the AWS Region of the data store.
resources	array (string)	A JSON array that contains the ARN of the data store.
detail	object	A JSON object that contains information about the event.

Name	Type	Description
detail.jobId	string	The import job ID associated with the status change event.
detail.submitTime	string	The time the import job was submitted.
detail.datastoreId	string	The data store that generated the status change event.
detail.inputDataConfig	string	The input prefix path for the Amazon S3 bucket that contains the FHIR files to be imported.

Export job events

Export Job Submitted

State - SUBMITTED

```
{
  "version": "0",
  "id": "f8af7d04-2221-4f02-a01a-6fc3ae403bab",
  "detail-type": "Export Job Submitted",
  "source": "aws.healthlake",
  "account": "123456789012",
  "time": "2023-12-08T01:50:51Z",
  "region": "us-east-1",
  "resources":
  [
    "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd6c68cf2dfd"
  ],
  "detail":
  {
    "jobId": "45e899e545bf774710388260fc60b143",
    "submitTime": "2023-12-08T01:50:50.986Z",
    "datastoreId": "eeb8005725ae22b35b4edbd6c68cf2dfd",
    "outputDataConfig":
```



```
{
  "s3Uri": "s3://amzn-s3-demo-source-bucket/output/"
}
```

Export Job In Progress

State - IN_PROGRESS

```
{
  "version": "0",
  "id": "7bb7e39c-707d-4a83-8532-cee015299100",
  "detail-type": "Export Job In Progress",
  "source": "aws.healthlake",
  "account": "123456789012",
  "time": "2023-12-08T01:51:23Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/eeb8005725ae22b35b4edbd6c68cf2dfd"
  ],
  "detail": {
    "jobId": "45e899e545bf774710388260fc60b143",
    "submitTime": "2023-12-08T01:50:50.986Z",
    "datastoreId": "eeb8005725ae22b35b4edbd6c68cf2dfd",
    "outputDataConfig": {
      "s3Uri": "s3://amzn-s3-demo-source-bucket/output/"
    }
  }
}
```

Export Job Completed

State - COMPLETED

```
{
  "version": "0",
  "id": "d7629aa7-e63a-4b84-858c-96a62b57ebc8",
  "detail-type": "Export Job Completed",
```

```

    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T02:14:42Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd6c68cf2dfd"
    ],
    "detail":
    {
        "jobId": "45e899e545bf774710388260fc60b143",
        "submitTime": "2023-12-08T01:50:50.986Z",
        "datastoreId": "eeb8005725ae22b35b4edbd6c68cf2dfd",
        "outputDataConfig":
        {
            "s3Uri": "s3://amzn-s3-demo-source-bucket/output/"
        }
    }
}

```

Export Job Completed With Errors

State - COMPLETED_WITH_ERRORS

```

{
    "version": "0",
    "id": "5fa50bc5-50e3-4bc4-b66a-1b1d2f7b07a7",
    "detail-type": "Export Job Completed With Errors",
    "source": "aws.healthlake",
    "account": "123456789012",
    "time": "2023-12-08T02:14:42Z",
    "region": "us-east-1",
    "resources":
    [
        "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/
eeb8005725ae22b35b4edbd6c68cf2dfd"
    ],
    "detail":
    {
        "jobId": "45e899e545bf774710388260fc60b143",
        "submitTime": "2023-12-08T01:50:50.986Z",
        "datastoreId": "eeb8005725ae22b35b4edbd6c68cf2dfd",
        "outputDataConfig":
    
```

```
    {
      "s3Uri": "s3://amzn-s3-demo-source-bucket/output/"
    }
  }
}
```

Export Job Failed

State - FAILED

```
{
  "version": "0",
  "id": "49fce45e-7e02-4846-8582-e7f19ca039cb",
  "detail-type": "Export Job Failed",
  "source": "aws.healthlake",
  "account": "123456789012",
  "time": "2023-12-08T02:14:42Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:healthlake:us-east-1:123456789012:datastore/fhir/eeb8005725ae22b35b4edbd68cf2dfd"
  ],
  "detail": {
    "jobId": "45e899e545bf774710388260fc60b143",
    "submitTime": "2023-12-08T01:50:50.986Z",
    "datastoreId": "eeb8005725ae22b35b4edbd68cf2dfd",
    "outputDataConfig": {
      "s3Uri": "s3://amzn-s3-demo-source-bucket/output/"
    }
  }
}
```

Export job events - metadata descriptions

Name	Type	Description
version	string	The EventBridge event schema version.

Name	Type	Description
id	string	The Version 4 UUID generated for every event.
detail-type	string	The type of event that is being sent.
source	string	Identifies the service that generated the event.
account	string	The 12-digit AWS account ID of the data store owner.
time	string	The time the event occurred.
region	string	Identifies the AWS Region of the data store.
resources	array (string)	A JSON array that contains the ARN of the data store.
detail	object	A JSON object that contains information about the event.
detail.jobId	string	The export job ID associated with the status change event.
detail.submitTime	string	The time the export job was submitted.
detail.datastoreId	string	The data store that generated the status change event.
detail.outputDataConfig	string	The output prefix path for the Amazon S3 bucket that contains the FHIR files to be exported.

Security in AWS HealthLake

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the [AWS Compliance Programs](#). To learn about the compliance programs that apply to HealthLake, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using HealthLake. The following topics show you how to configure HealthLake to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your HealthLake resources.

Topics

- [Data Protection in AWS HealthLake](#)
- [Encryption at REST for AWS HealthLake](#)
- [Encryption in transit for AWS HealthLake](#)
- [Identity and access management for AWS HealthLake](#)
- [Compliance validation for AWS HealthLake](#)
- [Infrastructure security in AWS HealthLake](#)
- [Creating AWS HealthLake resources with AWS CloudFormation](#)
- [AWS HealthLake and interface VPC endpoints \(AWS PrivateLink\)](#)
- [Security best practices in AWS HealthLake](#)
- [Resilience in AWS HealthLake](#)

Data Protection in AWS HealthLake

The AWS [shared responsibility model](#) applies to data protection in AWS HealthLake. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](#). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR](#) blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail. For information about using CloudTrail trails to capture AWS activities, see [Working with CloudTrail trails](#) in the *AWS CloudTrail User Guide*.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-3 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard \(FIPS\) 140-3](#).

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with HealthLake or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Encryption at REST for AWS HealthLake

HealthLake provides encryption by default to protect sensitive customer data at rest by using a service owned AWS Key Management Service (AWS KMS) key. Customer-managed KMS keys are also supported and are required for both importing and exporting files from a data store. To learn more about Customer-managed KMS Key, see [Amazon Key Management Service](#). Customers can choose an AWS owned KMS key or a Customer-managed KMS key when creating a data store. The encryption configuration cannot be changed after a data store has been created. If a data store is using an AWS owned KMS Key, it will be denoted as `AWS_OWNED_KMS_KEY` and you will not see the specific key used for encryption at rest.

AWS owned KMS key

HealthLake uses these keys by default to automatically encrypt potentially sensitive information such as personally identifiable or Private Health Information(PHI) data at rest. AWS owned KMS keys aren't stored in your account. They're part of a collection of KMS keys that AWS owns and manages for use in multiple AWS accounts. AWS services can use AWS owned KMS keys to protect your data. You can't view, manage, use AWS owned KMS keys, or audit their use. However, you don't need to do any work or change any programs to protect the keys that encrypt your data.

You're not charged a monthly fee or a usage fee if you use AWS owned KMS keys, and they don't count against AWS KMS quotas for your account. For more information, see [AWS owned keys](#).

Customer managed KMS keys

HealthLake supports the use of a symmetric customer managed KMS key that you create, own, and manage to add a second layer of encryption over the existing AWS owned encryption. Because you have full control of this layer of encryption, you can perform such tasks as:

- Establishing and maintaining key policies, IAM policies, and grants
- Rotating key cryptographic material
- Enabling and disabling key policies
- Adding tags
- Creating key aliases
- Scheduling keys for deletion

You can also use CloudTrail to track the requests that HealthLake sends to AWS KMS on your behalf. Additional AWS KMS charges apply. For more information, see [customer owned keys](#).

Create a customer managed key

You can create a symmetric customer managed key by using the AWS Management Console, or the AWS KMS APIs.

Follow the steps for [Creating symmetric customer managed key](#) in the AWS Key Management Service Developer Guide.

Key policies control access to your customer managed key. Every customer managed key must have exactly one key policy, which contains statements that determine who can use the key and how they can use it. When you create your customer managed key, you can specify a key policy. For more information, see [Managing access to customer managed keys](#) in the AWS Key Management Service Developer Guide.

To use your customer managed key with your HealthLake resources, [kms:CreateGrant](#) operations must be permitted in the key policy. This adds a grant to a customer managed key which controls access to a specified KMS key, which gives a user access to the [kms:grant operations](#) HealthLake requires. See [Using grants](#) for more information.

To use your customer managed KMS key with your HealthLake resources, the following API operations must be permitted in the key policy:

- `kms:CreateGrant` adds grants to a specific customer managed KMS key which allows access to grant operations.
- `kms:DescribeKey` provides the customer managed key details needed to validate the key. This is required for all operations.
- `kms:GenerateDataKey` provides access to encrypt resources at rest for all write operations.
- `kms:Decrypt` provides access to read or search operations for encrypted resources.

The following is a policy statement example that allows a user to create and interact with a data store in AWS HealthLake which is encrypted by that key:

```
"Statement": [
```



```

{
  "Sid": "Allow access to create data stores and do CRUD/search in AWS
HealthLake",
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::111122223333:HealthLakeFullAccessRole"
  },
  "Action": [
    "kms:DescribeKey",
    "kms:CreateGrant",
    "kms:GenerateDataKey",
    "kms:Decrypt"
  ],
  "Resource": "*",
  "Condition": {
    "StringEquals": {
      "kms:ViaService": "healthlake.amazonaws.com",
      "kms:CallerAccount": "111122223333"
    }
  }
}
]

```

Required IAM permissions for using a customer managed KMS key

When creating a data store with AWS KMS encryption enabled using a customer managed KMS key, there are required permissions for both the key policy and the IAM policy for the user or role creating the HealthLake data store.

You can use the [kms:ViaService condition key](#) to limit use of the KMS key to only requests that originate from HealthLake.

For more information about key policies, see [Enabling IAM policies](#) in the AWS Key Management Service Developer Guide.

The IAM user, IAM role, or AWS account creating your repositories must have the `kms:CreateGrant`, `kms:GenerateDataKey`, and `kms:DescribeKey` permissions plus the necessary HealthLake permissions.

How HealthLake uses grants in AWS KMS

HealthLake requires a [grant](#) to use your customer managed KMS key. When you create a Data Store encrypted with a customer managed KMS key, HealthLake creates a grant on your behalf by sending a [CreateGrant](#) request to AWS KMS. Grants in AWS KMS are used to give HealthLake access to a KMS key in a customer account.

The grants that HealthLake creates on your behalf should not be revoked or retired. If you revoke or retire the grant that gives HealthLake permission to use the AWS KMS keys in your account, HealthLake cannot access this data, encrypt new FHIR resources pushed to the data store, or decrypt them when they are pulled. When you revoke or retire a grant for HealthLake, the change occurs immediately. To revoke access rights, you should delete the data store rather than revoking the grant. When a data store is deleted, HealthLake retires the grants on your behalf.

Monitoring your encryption keys for HealthLake

You can use CloudTrail to track the requests that HealthLake sends to AWS KMS on your behalf when using a customer managed KMS key. The log entries in the CloudTrail log show `healthlake.amazonaws.com` in the `userAgent` field to clearly distinguish requests made by HealthLake.

The following examples are CloudTrail events for `CreateGrant`, `GenerateDataKey`, `Decrypt`, and `DescribeKey` to monitor AWS KMS operations called by HealthLake to access data encrypted by your customer managed key.

The following shows how to use `CreateGrant` to allow HealthLake to access a customer provided KMS key, enabling HealthLake to use that KMS key to encrypt all customer data at rest.

Users are not required to create their own grants. HealthLake creates a grant on your behalf by sending a `CreateGrant` request to AWS KMS. Grants in AWS KMS are used to give HealthLake access to a AWS KMS key in a customer account.

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLEROLE:Sampleuser01",
    "arn": "arn:aws:sts::111122223333:assumed-role/Sampleuser01",
    "accountId": "111122223333",
    "accessKeyId": "EXAMPLEKEYID",
```

```

    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "EXAMPLEROLE",
        "arn": "arn:aws:iam::111122223333:role/Sampleuser01",
        "accountId": "111122223333",
        "userName": "Sampleuser01"
      },
      "webIdFederationData": {},
      "attributes": {
        "creationDate": "2021-06-30T19:33:37Z",
        "mfaAuthenticated": "false"
      }
    },
    "invokedBy": "healthlake.amazonaws.com"
  },
  "eventTime": "2021-06-30T20:31:15Z",
  "eventSource": "kms.amazonaws.com",
  "eventName": "CreateGrant",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "healthlake.amazonaws.com",
  "userAgent": "healthlake.amazonaws.com",
  "requestParameters": {
    "operations": [
      "CreateGrant",
      "Decrypt",
      "DescribeKey",
      "Encrypt",
      "GenerateDataKey",
      "GenerateDataKeyWithoutPlaintext",
      "ReEncryptFrom",
      "ReEncryptTo",
      "RetireGrant"
    ],
    "granteePrincipal": "healthlake.us-east-1.amazonaws.com",
    "keyId": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN",
    "retiringPrincipal": "healthlake.us-east-1.amazonaws.com"
  },
  "responseElements": {
    "grantId": "EXAMPLE_ID_01"
  },
  "requestID": "EXAMPLE_ID_02",
  "eventID": "EXAMPLE_ID_03",
  "readOnly": false,

```

```

    "resources": [
      {
        "accountId": "111122223333",
        "type": "AWS::KMS::Key",
        "ARN": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN"
      }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
  }

```

The following examples shows how to use `GenerateDataKey` to ensure the user has necessary permissions to encrypt data before storing it.

```

    {
      "eventVersion": "1.08",
      "userIdentity": {
        "type": "AssumedRole",
        "principalId": "EXAMPLEUSER",
        "arn": "arn:aws:sts::111122223333:assumed-role/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "EXAMPLEKEYID",
        "sessionContext": {
          "sessionIssuer": {
            "type": "Role",
            "principalId": "EXAMPLEROLE",
            "arn": "arn:aws:iam::111122223333:role/Sampleuser01",
            "accountId": "111122223333",
            "userName": "Sampleuser01"
          },
          "webIdFederationData": {},
          "attributes": {
            "creationDate": "2021-06-30T21:17:06Z",
            "mfaAuthenticated": "false"
          }
        }
      },
      "invokedBy": "healthlake.amazonaws.com"
    },
    "eventTime": "2021-06-30T21:17:37Z",

```

```

    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "healthlake.amazonaws.com",
    "userAgent": "healthlake.amazonaws.com",
    "requestParameters": {
        "keySpec": "AES_256",
        "keyId": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN"
    },
    "responseElements": null,
    "requestID": "EXAMPLE_ID_01",
    "eventID": "EXAMPLE_ID_02",
    "readOnly": true,
    "resources": [
        {
            "accountId": "111122223333",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management"
}

```

The following example shows how HealthLake calls the Decrypt operation to use the stored encrypted data key to access the encrypted data.

```

{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "EXAMPLEUSER",
        "arn": "arn:aws:sts::111122223333:assumed-role/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "EXAMPLEKEYID",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "EXAMPLEROLE",

```

```

        "arn": "arn:aws:iam::111122223333:role/Sampleuser01",
        "accountId": "111122223333",
        "userName": "Sampleuser01"
    },
    "webIdFederationData": {},
    "attributes": {
        "creationDate": "2021-06-30T21:17:06Z",
        "mfaAuthenticated": "false"
    }
},
"invokedBy": "healthlake.amazonaws.com"
},
"eventTime": "2021-06-30T21:21:59Z",
"eventSource": "kms.amazonaws.com",
"eventName": "Decrypt",
"awsRegion": "us-east-1",
"sourceIPAddress": "healthlake.amazonaws.com",
"userAgent": "healthlake.amazonaws.com",
"requestParameters": {
    "encryptionAlgorithm": "SYMMETRIC_DEFAULT",
    "keyId": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN"
},
"responseElements": null,
"requestID": "EXAMPLE_ID_01",
"eventID": "EXAMPLE_ID_02",
"readOnly": true,
"resources": [
    {
        "accountId": "111122223333",
        "type": "AWS::KMS::Key",
        "ARN": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN"
    }
],
"eventType": "AwsApiCall",
"managementEvent": true,
"recipientAccountId": "111122223333",
"eventCategory": "Management"
}

```

The following example shows how HealthLake uses the DescribeKey operation to verify if the AWS KMS customer owned AWS KMS key is in a usable state and to help a user troubleshoot if it is not functional.

```

    {
      "eventVersion": "1.08",
      "userIdentity": {
        "type": "AssumedRole",
        "principalId": "EXAMPLEUSER",
        "arn": "arn:aws:sts::111122223333:assumed-role/Sampleuser01",
        "accountId": "111122223333",
        "accessKeyId": "EXAMPLEKEYID",
        "sessionContext": {
          "sessionIssuer": {
            "type": "Role",
            "principalId": "EXAMPLEROLE",
            "arn": "arn:aws:iam::111122223333:role/Sampleuser01",
            "accountId": "111122223333",
            "userName": "Sampleuser01"
          },
          "webIdFederationData": {},
          "attributes": {
            "creationDate": "2021-07-01T18:36:14Z",
            "mfaAuthenticated": "false"
          }
        }
      },
      "invokedBy": "healthlake.amazonaws.com"
    },
    "eventTime": "2021-07-01T18:36:36Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
    "awsRegion": "us-east-1",
    "sourceIPAddress": "healthlake.amazonaws.com",
    "userAgent": "healthlake.amazonaws.com",
    "requestParameters": {
      "keyId": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN"
    },
    "responseElements": null,
    "requestID": "EXAMPLE_ID_01",
    "eventID": "EXAMPLE_ID_02",
    "readOnly": true,
    "resources": [
      {
        "accountId": "111122223333",
        "type": "AWS::KMS::Key",
        "ARN": "arn:aws:kms:us-east-1:111122223333:key/EXAMPLE_KEY_ARN"
      }
    ]
  }

```

```
    }  
  ],  
  "eventType": "AwsApiCall",  
  "managementEvent": true,  
  "recipientAccountId": "111122223333",  
  "eventCategory": "Management"  
}
```

Learn more

The following resources provide more information about data at rest encryption.

For more information about [AWS Key Management Service basic concepts](#), see the AWS KMS documentation.

For more information about [Security best practices](#) in the AWS KMS documentation.

Encryption in transit for AWS HealthLake

AWS HealthLake uses TLS 1.2 to encrypt data in transit through the public endpoint and through backend services.

Identity and access management for AWS HealthLake

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use HealthLake resources. IAM is an AWS service that you can use with no additional charge.

Topics

- [Audience](#)
- [Authenticating with identities](#)
- [Managing access using policies](#)
- [How AWS HealthLake works with IAM](#)
- [Identity-based policy examples for AWS HealthLake](#)
- [AWS managed policies for AWS HealthLake](#)
- [Troubleshooting AWS HealthLake identity and access](#)

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in HealthLake.

Service user – If you use the HealthLake service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more HealthLake features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in HealthLake, see [Troubleshooting AWS HealthLake identity and access](#).

Service administrator – If you're in charge of HealthLake resources at your company, you probably have full access to HealthLake. It's your job to determine which HealthLake features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with HealthLake, see [How AWS HealthLake works with IAM](#).

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to HealthLake. To view example HealthLake identity-based policies that you can use in IAM, see [Identity-based policy examples for AWS HealthLake](#).

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see [How to sign in to your AWS account](#) in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If

you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see [AWS Signature Version 4 for API requests](#) in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see [Multi-factor authentication](#) in the *AWS IAM Identity Center User Guide* and [AWS Multi-factor authentication in IAM](#) in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see [Tasks that require root user credentials](#) in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see [What is IAM Identity Center?](#) in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An [IAM user](#) is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating

IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see [Rotate access keys regularly for use cases that require long-term credentials](#) in the *IAM User Guide*.

An [IAM group](#) is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see [Use cases for IAM users](#) in the *IAM User Guide*.

IAM roles

An [IAM role](#) is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. To temporarily assume an IAM role in the AWS Management Console, you can [switch from a user to an IAM role \(console\)](#). You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see [Methods to assume a role](#) in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see [Create a role for a third-party identity provider \(federation\)](#) in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see [Permission sets](#) in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource

(instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
- **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).
- **Service role** – A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Create a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.
- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see [Use an IAM role to grant permissions to applications running on Amazon EC2 instances](#) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their

permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see [Overview of JSON policies](#) in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Define custom IAM permissions with customer managed policies](#) in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see [Choose between managed policies and inline policies](#) in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#) in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list \(ACL\) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [Service control policies](#) in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – RCPs are JSON policies that you can use to set the maximum available permissions for resources in your accounts without updating the IAM policies attached to each resource that you own. The RCP limits permissions for resources in member accounts and can impact the effective permissions for identities, including the AWS account root user, regardless of whether they belong to your organization. For more information about

Organizations and RCPs, including a list of AWS services that support RCPs, see [Resource control policies \(RCPs\)](#) in the *AWS Organizations User Guide*.

- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

How AWS HealthLake works with IAM

Before you use IAM to manage access to HealthLake, learn what IAM features are available to use with HealthLake.

IAM features you can use with AWS HealthLake

IAM feature	HealthLake support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Yes
Temporary credentials	Yes
Principal permissions	Yes

IAM feature	HealthLake support
Service roles	Yes
Service-linked roles	No

To get a high-level view of how HealthLake and other AWS services work with most IAM features, see [AWS services that work with IAM](#) in the *IAM User Guide*.

Identity-based policies for AWS HealthLake

Supports identity-based policies: Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see [Define custom IAM permissions with customer managed policies](#) in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see [IAM JSON policy elements reference](#) in the *IAM User Guide*.

Identity-based policy examples for AWS HealthLake

To view examples of HealthLake identity-based policies, see [Identity-based policy examples for AWS HealthLake](#).

Resource-based policies within AWS HealthLake

Supports resource-based policies: No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must [specify a principal](#)

in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

Policy actions for AWS HealthLake

Supports policy actions: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The **Action** element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of HealthLake actions, see [Actions defined by AWS HealthLake](#) in the *Service Authorization Reference*.

Policy actions in HealthLake use the following prefix before the action:

```
healthlake
```

To specify multiple actions in a single statement, separate each action with a comma.

```
"Action": [  
    "healthlake:action1",
```

```
"healthlake:action2"  
]
```

To view examples of HealthLake identity-based policies, see [Identity-based policy examples for AWS HealthLake](#).

Policy resources for AWS HealthLake

Supports policy resources: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The Resource JSON policy element specifies the object or objects to which the action applies. Statements must include either a Resource or a NotResource element. As a best practice, specify a resource using its [Amazon Resource Name \(ARN\)](#). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of HealthLake resource types and their ARNs, see [Resources defined by AWS HealthLake](#) in the *Service Authorization Reference*. To learn the actions with which you can specify the ARN of each resource, see [Actions defined by AWS HealthLake](#).

To view examples of HealthLake identity-based policies, see [Identity-based policy examples for AWS HealthLake](#).

Policy condition keys for AWS HealthLake

Supports service-specific policy condition keys: Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or *Condition block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use [condition operators](#), such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical AND operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical OR operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see [IAM policy elements: variables and tags](#) in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see [AWS global condition context keys](#) in the *IAM User Guide*.

To see a list of HealthLake condition keys, see [Condition keys for AWS HealthLake](#) in the *Service Authorization Reference*. To learn the actions and resources with which you can use a condition key, see [Actions defined by AWS HealthLake](#).

To view examples of HealthLake identity-based policies, see [Identity-based policy examples for AWS HealthLake](#).

Access control lists (ACLs) in AWS HealthLake

Supports ACLs: No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Attribute-based access control (ABAC) with AWS HealthLake

Supports ABAC (tags in policies): Yes

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the [condition element](#) of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see [Define permissions with ABAC authorization](#) in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see [Use attribute-based access control \(ABAC\)](#) in the *IAM User Guide*.

Using temporary credentials with AWS HealthLake

Supports temporary credentials: Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see [AWS services that work with IAM](#) in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see [Switch from a user to an IAM role \(console\)](#) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see [Temporary security credentials in IAM](#).

Cross-service principal permissions for AWS HealthLake

Supports forward access sessions (FAS): Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a

different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see [Forward access sessions](#).

Service roles for AWS HealthLake

Supports service roles: Yes

A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Create a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

For information about service roles and the inline policy required for full access to AWS HealthLake, see [Setting up AWS HealthLake](#).

Warning

Changing the permissions for a service role might break HealthLake functionality. Edit service roles only when HealthLake provides guidance to do so.

Service-linked roles for AWS HealthLake

Supports service-linked roles: No

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see [AWS services that work with IAM](#). Find a service in the table that includes a Yes in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for AWS HealthLake

By default, users and roles don't have permission to create or modify HealthLake resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface

(AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see [Create IAM policies \(console\)](#) in the *IAM User Guide*.

For details about actions and resource types defined by HealthLake, including the format of the ARNs for each of the resource types, see [Actions, resources, and condition keys for AWS HealthLake](#) in the *Service Authorization Reference*.

Topics

- [Policy best practices](#)
- [Using the AWS HealthLake console](#)
- [Accessing an AWS HealthLake data store in Amazon Athena](#)
- [Allowing users to view their own permissions](#)

Policy best practices

Identity-based policies determine whether someone can create, access, or delete HealthLake resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see [AWS managed policies](#) or [AWS managed policies for job functions](#) in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see [Policies and permissions in IAM](#) in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to

specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see [IAM JSON policy elements: Condition](#) in the *IAM User Guide*.

- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see [Validate policies with IAM Access Analyzer](#) in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see [Secure API access with MFA](#) in the *IAM User Guide*.

For more information about best practices in IAM, see [Security best practices in IAM](#) in the *IAM User Guide*.

Using the AWS HealthLake console

To access the AWS HealthLake console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the HealthLake resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

For full access to HealthLake, attach the following policies to an IAM user or role:

AmazonHealthLakeFullAccess and AWSLakeFormationDataAdmin. You also need to attach the HealthLake inline policy which is a service role. A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see [Create a role to delegate permissions to an AWS service](#) in the *IAM User Guide*. For information about the inline policy which creates the required service role, see [Setting up AWS HealthLake](#). You must also use the AWS Lake Formation console or CLI to assign your HealthLake administrator to be an AWS Lake Formation Data Lake administrator. For more information, see [Setting up AWS HealthLake](#).

Accessing an AWS HealthLake data store in Amazon Athena

If you want to provide users and roles with access to the HealthLake data stores in Amazon Athena, attach the following IAM policies to the role or user: `AmazonAthenaFullAccess` and `AmazonS3FullAccess`. `Select` and `Describe` permissions are also required on tables managed by AWS Lake Formation. AWS Lake Formation table permissions are granted by an AWS Lake Formation administrator in the AWS Lake Formation console or via the CLI. For more information, see [Setting up AWS HealthLake](#)

Allowing users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ViewOwnUserInfo",
      "Effect": "Allow",
      "Action": [
        "iam:GetUserPolicy",
        "iam:ListGroupsForUser",
        "iam:ListAttachedUserPolicies",
        "iam:ListUserPolicies",
        "iam:GetUser"
      ],
      "Resource": ["arn:aws:iam::*:user/${aws:username}"]
    },
    {
      "Sid": "NavigateInConsole",
      "Effect": "Allow",
      "Action": [
        "iam:GetGroupPolicy",
        "iam:GetPolicyVersion",
        "iam:GetPolicy",
        "iam:ListAttachedGroupPolicies",
        "iam:ListGroupPolicies",
        "iam:ListPolicyVersions",
        "iam:ListPolicies",
        "iam:ListUsers"
      ]
    }
  ]
}
```



```
    ],  
    "Resource": "*"    
  }  
]  
}
```

AWS managed policies for AWS HealthLake

An AWS managed policy is a standalone policy that is created and administered by AWS. AWS managed policies are designed to provide permissions for many common use cases so that you can start assigning permissions to users, groups, and roles.

Keep in mind that AWS managed policies might not grant least-privilege permissions for your specific use cases because they're available for all AWS customers to use. We recommend that you reduce permissions further by defining [customer managed policies](#) that are specific to your use cases.

You cannot change the permissions defined in AWS managed policies. If AWS updates the permissions defined in an AWS managed policy, the update affects all principal identities (users, groups, and roles) that the policy is attached to. AWS is most likely to update an AWS managed policy when a new AWS service is launched or new API operations become available for existing services.

For more information, see [AWS managed policies](#) in the *IAM User Guide*.

AWS managed policy: AmazonHealthLakeFullAccess

The AmazonHealthLakeFullAccess policy provides full access to HealthLake. With this policy attached to their user or role, users can use HealthLake to access, query, import, and export data in HealthLake. To perform many common actions in HealthLake, you must add additional policies to the user or role. For more information, see [Setting up AWS HealthLake](#) and [HealthLake operations and permissions](#).

You can attach the `AmazonHealthLakeFullAccess` policy to your IAM identities.

This policy grants administrative and contributor permissions that allow users and roles to query, search, import, and export with HealthLake, and it also makes it possible for HealthLake to perform actions on behalf of the users and roles that have these permissions.

Permissions details

This policy includes the following statement.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "healthlake:*",
        "s3:ListAllMyBuckets",
        "s3:ListBucket",
        "s3:GetBucketLocation",
        "iam:ListRoles"
      ],
      "Resource": "*",
      "Effect": "Allow"
    },
    {
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "iam:PassedToService": "healthlake.amazonaws.com"
        }
      }
    }
  ]
}
```

AWS managed policy: AmazonHealthLakeReadOnlyAccess

AmazonHealthLakeReadOnlyAccess policy grants read-only access and permissions to HealthLake and related resources in other AWS services. Apply this policy to users who you want to grant the ability to query and view HealthLake data store, but not the ability to create or make changes to them.

You can attach the AmazonHealthLakeReadOnlyAccess policy to your IAM identities.

This policy grants *read-only* permissions that allow users and roles to query HealthLake.

Permissions details

This policy includes the following statement.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": [
        "healthlake:ListFHIRDatastores",
        "healthlake:DescribeFHIRDatastore",
        "healthlake:DescribeFHIRImportJob",
        "healthlake:DescribeFHIRExportJob",
        "healthlake:GetCapabilities",
        "healthlake:ReadResource",
        "healthlake:SearchWithGet",
        "healthlake:SearchWithPost",
        "healthlake:SearchEverything"
      ],
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

HealthLake operations and permissions

The following table lists typical operations in HealthLake and the permissions needed to perform them.

HealthLake operations	Required permissions
Create a data store in HealthLake	AmazonHealthLakeFullAccess , AmazonLakeFormationDataAdmin , inline policy , and AWS Lake Formation Administrator permissions managed by AWS Lake Formation
Delete a data store in HealthLake	AmazonHealthLakeFullAccess , AmazonLakeFormationDataAdmin , inline policy , and AWS Lake Formation Administrator permissions managed by AWS Lake Formation
List, search, or query a data store in HealthLake	AmazonHealthLakeReadOnlyAccess
Query a data store using Amazon Athena	AmazonAthenaFullAccess , AmazonS3FullAccess , AWS Lake Formation Select and Describe permissions on tables managed by AWS Lake Formation
Import data from HealthLake	See Setting up permissions for import jobs .
Export data from HealthLake	See Setting up permissions for export jobs .

HealthLake updates to AWS managed policies

View details about updates to AWS managed policies for HealthLake from the time that this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the HealthLake Document history page.

Change	Description	Date
AmazonHealthLakeFullAccess	AmazonHealthLakeFullAccess policy required to allow full access to HealthLake.	November, 14, 2022
AmazonHealthLakeReadOnlyAccess	AmazonHealthLakeReadOnlyAccess policy required for read-only access to HealthLake.	November, 14, 2022
HealthLake started tracking changes	HealthLake started tracking changes for its AWS managed policies.	November, 14, 2022

Troubleshooting AWS HealthLake identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with HealthLake and IAM.

Topics

- [I am not authorized to perform an action in AWS HealthLake](#)
- [I am not authorized to perform iam:PassRole](#)
- [I want to allow people outside of my AWS account to access my AWS HealthLake resources](#)

I am not authorized to perform an action in AWS HealthLake

If the AWS Management Console tells you that you're not authorized to perform an action, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password.

The following example error occurs when the mateojackson IAM user tries to use the console to view details about a fictional *my-example-widget* resource but does not have the fictional `healthlake:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
healthlake:GetWidget on resource: my-example-widget
```

In this case, Mateo asks his administrator to update his policies to allow him to access the *my-example-widget* resource using the healthlake: *GetWidget* action.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the iam:PassRole action, your policies must be updated to allow you to pass a role to HealthLake.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named marymajor tries to use the console to perform an action in HealthLake. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the iam:PassRole action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my AWS HealthLake resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether HealthLake supports these features, see [How AWS HealthLake works with IAM](#).
- To learn how to provide access to your resources across AWS accounts that you own, see [Providing access to an IAM user in another AWS account that you own](#) in the *IAM User Guide*.

- To learn how to provide access to your resources to third-party AWS accounts, see [Providing access to AWS accounts owned by third parties](#) in the *IAM User Guide*.
- To learn how to provide access through identity federation, see [Providing access to externally authenticated users \(identity federation\)](#) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see [Cross account resource access in IAM](#) in the *IAM User Guide*.

Compliance validation for AWS HealthLake

Third-party auditors assess the security and compliance of AWS HealthLake as part of multiple AWS compliance programs. For HealthLake this includes HIPAA.

To learn whether an AWS service is within the scope of specific compliance programs, see [AWS services in Scope by Compliance Program](#) and choose the compliance program that you are interested in. For general information, see [AWS Compliance Programs](#).

You can download third-party audit reports using AWS Artifact. For more information, see [Downloading Reports in AWS Artifact](#).

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- [Security Compliance & Governance](#) – These solution implementation guides discuss architectural considerations and provide steps for deploying security and compliance features.
- [HIPAA Eligible Services Reference](#) – Lists HIPAA eligible services. Not all AWS services are HIPAA eligible.
- [AWS Compliance Resources](#) – This collection of workbooks and guides might apply to your industry and location.
- [AWS Customer Compliance Guides](#) – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- [Evaluating Resources with Rules](#) in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.

- [AWS Security Hub](#) – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see [Security Hub controls reference](#).
- [Amazon GuardDuty](#) – This AWS service detects potential threats to your AWS accounts, workloads, containers, and data by monitoring your environment for suspicious and malicious activities. GuardDuty can help you address various compliance requirements, like PCI DSS, by meeting intrusion detection requirements mandated by certain compliance frameworks.
- [AWS Audit Manager](#) – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Infrastructure security in AWS HealthLake

As a managed service, AWS HealthLake is protected by the AWS global network security procedures that are described in the [Amazon Web Services: Overview of Security Processes](#) whitepaper.

You use AWS published API calls to access HealthLake through the network. Clients must support Transport Layer Security (TLS) 1.0 or later. We recommend TLS 1.2 or later. Clients must also support cipher suites with perfect forward secrecy (PFS) such as Ephemeral Diffie-Hellman (DHE) or Elliptic Curve Ephemeral Diffie-Hellman (ECDHE). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

Creating AWS HealthLake resources with AWS CloudFormation

AWS HealthLake is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want and AWS CloudFormation provisions and configures those resources for you.

When you use AWS CloudFormation, you can reuse your template to set up your HealthLake resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

HealthLake and AWS CloudFormation templates

To provision and configure resources for HealthLake and related services, you must understand [AWS CloudFormation templates](#). Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use AWS CloudFormation Designer to help you get started with AWS CloudFormation templates. For more information, see [What is AWS CloudFormation Designer?](#) in the *AWS CloudFormation User Guide*.

Note

AWS HealthLake supports creating data stores with AWS CloudFormation. For more information, including examples of JSON and YAML templates for provisioning HealthLake data stores, see the [AWS HealthLake resource type reference](#) in the *AWS CloudFormation User Guide*.

Learn more about AWS CloudFormation

To learn more about AWS CloudFormation, see the following resources:

- [AWS CloudFormation](#)
- [AWS CloudFormation User Guide](#)
- [AWS CloudFormation API Reference](#)
- [AWS CloudFormation Command Line Interface User Guide](#)

AWS HealthLake and interface VPC endpoints (AWS PrivateLink)

You can establish a private connection between your VPC and AWS HealthLake by creating an *interface VPC endpoint*. Interface VPC endpoints are powered by [AWS PrivateLink](#), a technology that you can use to privately access HealthLake; APIs without an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to communicate with HealthLake; APIs. Traffic between your VPC and HealthLake; does not leave the Amazon network.

Each interface endpoint is represented by one or more [Elastic Network Interfaces](#) in your subnets.

For more information, see [Interface VPC endpoints \(AWS PrivateLink\)](#) in the *Amazon VPC User Guide*.

Considerations for HealthLake VPC endpoints

Before you set up an interface VPC endpoint for HealthLake, be sure you review [Interface endpoint properties and limitations](#) in the *Amazon VPC User Guide*.

HealthLake supports making calls to all of its API actions from your VPC.

Creating an interface VPC endpoint for HealthLake;

You can create a VPC endpoint for the HealthLake; service using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see [Creating an interface endpoint](#) in the *Amazon VPC User Guide*.

Create a VPC endpoint for HealthLake; using the following service name:

- `com.amazonaws.region.healthlake`

If you turn on private DNS for the endpoint, you can make API requests to HealthLake using its default DNS name for the Region. For example, `healthlake.us-east-1.amazonaws.com`.

For more information, see [Accessing a service through an interface endpoint](#) in the *Amazon VPC User Guide*.

Creating a VPC endpoint policy for HealthLake

You can attach an endpoint policy to your VPC endpoint that controls access to HealthLake. The policy specifies the following information:

- The principal that can perform actions.
- The actions that can be performed.
- The resources on which actions can be performed.

For more information, see [Controlling access to services with VPC endpoints](#) in the *Amazon VPC User Guide*.

Example: VPC endpoint policy for HealthLake actions

The following is an example of an endpoint policy for HealthLake. When attached to an endpoint, this policy grants access to the HealthLake `CreateFHIRDatastore` action for all principals on all resources.

```
{
  "Statement": [
    {
      "Principal": "*",
      "Effect": "Allow",
      "Action": [
        "healthlake:create-fhir-datastore"
      ],
      "Resource": "*"
    }
  ]
}
```

Security best practices in AWS HealthLake

AWS HealthLake provides a number of security features to consider as you develop and implement your own security policies. The following best practices are general guidelines and don't represent a complete security solution. Because these best practices might not be appropriate or sufficient for your environment, treat them as helpful considerations rather than prescriptions.

- Implement least privilege access.
- Whenever possible, use Customer-Managed-Keys(CMKs) to encrypt your data. To learn more about CMKs, see [Amazon Key Management Service](#).
- Use Search with POST, not Search with GET when querying for PHI or PII in your data store.
- Limit access to sensitive and important auditing functions.
- When creating resources through the update or bulk import APIs, do not use PHI or PII, including the names of data stores and jobs, in any visible fields or in the logical FHIR ID (LID).
- When sending create, read, update, delete, or search requests, do not use PHI in the HTTP header.
- Enable AWS CloudTrail to audit AWS HealthLake use and to ensure that there is no unexpected activity.
- Review best practices for using Amazon S3 buckets securely. To learn more, see [Security best practices](#) in the *Amazon S3 user guide*.

Resilience in AWS HealthLake

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

If you need to replicate your data or applications over greater geographic distances, use AWS Local Regions. An AWS Local Region is a single data center designed to complement an existing AWS Region. Like all AWS Regions, AWS Local Regions are completely isolated from other AWS Regions.

For more information about AWS Regions and Availability Zones, see [AWS Global Infrastructure](#).

AWS HealthLake reference

The following supporting reference material is available for SMART on FHIR, FHIR, and AWS HealthLake.

Note

All native HealthLake actions and data types are described in a separate reference. For more information, see the [AWS HealthLake API Reference](#).

Topics

- [SMART on FHIR support for AWS HealthLake](#)
- [FHIR R4 support for AWS HealthLake](#)
- [Support reference for AWS HealthLake](#)

SMART on FHIR support for AWS HealthLake

A Substitutable Medical Applications and Reusable Technologies (SMART) on FHIR enabled HealthLake data store allows access to SMART on FHIR compliant applications. HealthLake data is accessed by authenticating and authorizing requests using a third-party authorization server. So instead of managing user credentials via AWS Identity and Access Management, you are doing so using a SMART on FHIR compliant authorization server.

Note

HealthLake supports SMART on FHIR versions 1.0 and 2.0. To learn more about these frameworks, see [SMART App Launch](#) in the *FHIR R4 documentation*.

HealthLake data stores support the following authentication and authorization frameworks for SMART on FHIR requests:

- **OpenID (AuthN):** for authenticating the person or client application is who (or what) they claim to be.

- **OAuth 2.0 (AuthZ):** for authorizing which FHIR resources in your HealthLake data store an authenticated request can read or write to. This is defined by the scopes set up in your authorization server.

You can create a SMART on FHIR enabled data store using the AWS CLI or AWS SDKs. For more information, see [Creating a HealthLake data store](#).

Topics

- [Getting started with SMART on FHIR](#)
- [HealthLake authentication requirements for SMART on FHIR](#)
- [SMART on FHIR OAuth 2.0 scopes supported by HealthLake](#)
- [Token validation using AWS Lambda](#)
- [Using fine-grained authorization with a SMART on FHIR enabled HealthLake data store](#)
- [Fetching the SMART on FHIR Discovery Document](#)
- [Making a FHIR REST API request on a SMART-enabled HealthLake data store](#)

Getting started with SMART on FHIR

The following topics describe how to get started with SMART on FHIR authorization for AWS HealthLake. They include the resources you must provision in your AWS account, the creation of a SMART on FHIR enabled HealthLake data store, and an example of how a SMART on FHIR client application interacts with an authorization server and a HealthLake data store.

Topics

- [Setting up resources for SMART on FHIR](#)
- [Client application workflow for SMART on FHIR](#)

Setting up resources for SMART on FHIR

The following steps define how SMART on FHIR requests are handled by HealthLake and the resources needed for them to succeed. The following elements work together in a workflow to make a SMART on FHIR request:

- **The end-user:** Generally, a patient or clinician using a third-party SMART on FHIR application to access data in a HealthLake data store.
- **The SMART on FHIR application (referred to as the client application):** An application that wants to access data found in HealthLake data store.
- **The authorization server:** An OpenID Connect compliant server that is able to authenticate users and issue access tokens.
- **The HealthLake data store:** A SMART on FHIR enabled HealthLake data store that uses a Lambda function to respond to FHIR REST requests which provide a bearer token.

For these elements to work together, you must create the following resources.

Note

We recommend creating your SMART on FHIR enabled HealthLake data store after you've set up the authorization server, defined the necessary [scopes](#) on it, and created a AWS Lambda function to handle [token](#) introspection.

1. Set up an authorization server endpoint

To use the SMART on FHIR framework you need to set up an third-party authorization server that can validate FHIR REST requests made on a data store. For more information, see [HealthLake authentication requirements for SMART on FHIR](#).

2. Define scopes on your authorization server to control HealthLake data store access levels

The SMART on FHIR framework uses OAuth scopes to determine what FHIR resources an authenticated request has access to and to what extent. Defining scopes are a way to design for least-privilege. For more information, see [SMART on FHIR OAuth 2.0 scopes supported by HealthLake](#).

3. Set up an AWS Lambda function capable of performing token introspection

A FHIR REST request sent by the client application on a SMART on FHIR enabled data store contains a JSON Web Token (JWT). For more information, see [Decoding a JWT](#).

4. Create a SMART on FHIR enabled HealthLake data store

To create a SMART on FHIR HealthLake data store you need to provide an `IdentityProviderConfiguration`. For more information, see [Creating a HealthLake data store](#).

Client application workflow for SMART on FHIR

The following section explains how to launch a client application and make a successful FHIR REST request on an HealthLake data store within the context of SMART on FHIR.

1. Make a GET request to Well-Known Uniform Resource Identifier using client application

A SMART enabled client application must make a GET request to find the authorization endpoints of your HealthLake data store. This is done via a Well-Known Uniform Resource Identifier (URI) request. For more information, see [Fetching the SMART on FHIR Discovery Document](#).

2. Request access and scopes

The client application uses the authorization endpoint of the authorization server, so that the user can login. This process authenticates the user. Scopes are used to define what FHIR resources in your HealthLake data store a client application can access. For more information, see [SMART on FHIR OAuth 2.0 scopes supported by HealthLake](#).

3. Access tokens

Now that the user has been authenticated, a client application receives a JWT access token from the authorization server. This token is provided when the client application sends a FHIR REST request to HealthLake. For more information, see [Token validation](#).

4. Make a FHIR REST API request on SMART on FHIR enabled HealthLake data store

The client application can now send a FHIR REST API request to a HealthLake data store endpoint using the access token provided by the authorization server. For more information, see [Making a FHIR REST API request on a SMART-enabled HealthLake data store](#).

5. Validate the JWT access token

To validate the access token sent in the FHIR REST request, use a Lambda function. For more information, see [Token validation using AWS Lambda](#).

HealthLake authentication requirements for SMART on FHIR

To access FHIR resources in a SMART on FHIR-enabled HealthLake data store, a client application must be authorized by an OAuth 2.0-compliant authorization server and present an OAuth Bearer token as part of a FHIR REST API request. To find the authorization server's endpoint, use the HealthLake SMART on FHIR Discovery Document via a Well-Known Uniform Resource Identifier. To learn more about this process, see [Fetching the SMART on FHIR Discovery Document](#).

When you create a SMART on FHIR HealthLake data store, you must define the authorization server's end point and the token endpoint in the metadata element of the `CreateFHIRDatastore` request. To learn more about defining the metadata element, see [Creating a HealthLake data store](#).

Using the authorization server endpoints, the client application will authenticate a user with the authorization service. Once authorized and authenticated, a JSON Web Token (JWT) is generated by the authorization service and passed to the client application. This token contains FHIR resource scopes that the client application is allowed to use, which in turn restricts what data the user is able to access. Optionally, if the launch scope was provided then the response will contain those details. To learn more about the SMART on FHIR scopes supported by HealthLake, see [SMART on FHIR OAuth 2.0 scopes supported by HealthLake](#).

Using the JWT granted by the authorization server, a client application makes FHIR REST API calls to a SMART on FHIR enabled HealthLake data store. To validate and decode the JWT, you need to create a Lambda function. HealthLake invokes this Lambda function on your behalf when a FHIR REST API request is received. To see an example starter Lambda function, see [Token validation using AWS Lambda](#).

Authorization server elements required to create a SMART on FHIR enabled HealthLake data store

In the `CreateFHIRDatastore` request, you need to provide the authorization endpoint and the token endpoint as part of the metadata element in the `IdentityProviderConfiguration` object. Both the authorization endpoint and token endpoint are required. To see example of how this is specified in `CreateFHIRDatastore` request, see [Creating a HealthLake data store](#).

Required claims to complete a FHIR REST API request on a SMART on FHIR enabled HealthLake data store

Your AWS Lambda function must contain the following claims for it to be a valid FHIR REST API request on a SMART on FHIR enabled HealthLake data store.

- **nbf:** [\(Not Before\) Claim](#) — The "nbf" (not before) claim identifies the time before which the JWT MUST NOT be accepted for processing. The processing of the "nbf" claim requires that the current date/time MUST be after or equal to the not-before date/time listed in the "nbf" claim. The sample Lambda function we provide converts `iat` from the server response into `nbf`.
- **exp:** [\(Expiration Time\) Claim](#) — The "exp" (expiration time) claim identifies the expiration time on or after which the JWT must not be accepted for processing.
- **isAuthorized:** A boolean set to `True`. Indicates that request has been authorized on the authorization server.
- **aud:** [\(Audience\) Claim](#) — The "aud" (audience) claim identifies the recipients that the JWT is intended for. This must be a SMART on FHIR enabled HealthLake data store endpoint.
- **scope:** This must be at least one FHIR resource related scope. This scope is defined on your authorization server. To learn more about FHIR resource related scopes accepted by HealthLake, see [SMART on FHIR resource scopes for HealthLake](#).

SMART on FHIR OAuth 2.0 scopes supported by HealthLake

HealthLake uses OAuth 2.0 as an authorization protocol. Using this protocol on your authorization server allows you to define HealthLake data store permissions (create, read, update, delete, and search) for FHIR resources that a client application has access to.

The SMART on FHIR framework defines a set of scopes that can be requested from the authorization server. For example, a client application that is only designed to allow patients to view their lab results or view their contact details should only be *authorized* to request read scopes.

Note

HealthLake provides support for both SMART on FHIR V1 and V2 as described below. The SMART on FHIR [AuthorizationStrategy](#) is set to one of the following three values when your data store is created:

- **SMART_ON_FHIR_V1** – Support for only SMART on FHIR V1, which includes `read` (`read/search`) and `write` (`create/update/delete`) permissions.
- **SMART_ON_FHIR** – Support for both SMART on FHIR V1 and V2, which includes `create`, `read`, `update`, `delete`, and `search` permissions.
- **AWS_AUTH** – The default AWS HealthLake authorization strategy; not affiliated with SMART on FHIR.

Standalone launch scope

HealthLake supports the standalone launch mode scope `launch/patient`.

In standalone launch mode a client application requests access to patient's clinical data because the user and patient are not known to the client application. Thus, the client application's authorization request explicitly requests the patient scope be returned. After successful authentication, the authorization server issues a access token containing the requested launch patient scope. The needed patient context is provided alongside the access token in the authorization server's response.

Supported launch mode scopes

Scope	Description
<code>launch/patient</code>	A parameter in a OAuth 2.0 authorization request requesting that patient data be returned in the authorization response.

SMART on FHIR resource scopes for HealthLake

HealthLake defines three levels of SMART on FHIR resource scopes.

- **patient** scopes grant access to specific data about a single Patient.
- **user** scopes grant access to specific data that a user can access.
- **system** scopes grant access to all FHIR resources found in the HealthLake data store.

The following sections list the syntax for constructing FHIR resource scopes using either SMART on FHIR V1 or SMART on FHIR V2.

 Note

The SMART on FHIR authorization strategy is set when your data store is created. For more information, see [AuthorizationStrategy](#) in the *AWS HealthLake API Reference*.

SMART on FHIR V1 scopes supported by HealthLake

When using SMART on FHIR V1, the general syntax for constructing FHIR resource scopes for HealthLake follows. To view the entire URL path in the following example, scroll over the **Copy** button.

```
('patient' | 'user' | 'system') '/' (fhir-resource | '*') '.' ('read' | 'write' | '*')
```

SMART on FHIR v1 supported authorization scopes

Scope syntax	Example scope	Result
patient/(fhir-resource '*'). ('read' 'write' '*')	patient/AllergyIntolerance.*	The patient client application has instance-level read/write access to all recorded allergies.
user/(fhir-resource '*').('read' 'write' '*')	user/Observation.read	The user client application has instance-level read/write access to all recorded observations.
system/('read' 'write' '*')	system/*.*	The system client application has read/write access to all FHIR resource data.

SMART on FHIR V2 scopes supported by HealthLake

When using SMART on FHIR V2, the general syntax for constructing FHIR resource scopes for HealthLake follows. To view the entire URL path in the following example, scroll over the **Copy** button.

```
('patient' | 'user' | 'system') '/' (fhir-resource | '*') '.' ('c' | 'r' | 'u' | 'd' | 's')
```

 **Note**

To use SMART on FHIR V2, you must pass in the value [permission-v2](#) into the metadata capabilities string, which is a member of the [IdentityProviderConfiguration](#) data type.

HealthLake supports granular scopes. For more information, see [supported granular scopes](#) in the *FHIR US Core Implementation Guide*.

SMART on FHIR V2 supported authorization scopes

Scope syntax	Example V1 scope	Result
patient/Observation.rs	user/Obse rvation.read	Permission to read and search Observation resource for the current patient.
system/*.cruds	system/*.*	The system client application has full create/read/update/delete/search access to all FHIR resource data.

Token validation using AWS Lambda

When you create a HealthLake SMART on FHIR enabled data store, you must provide the ARN of the AWS Lambda function in the `CreateFHIRDatastore` request. The ARN of the Lambda function is specified in `IdentityProviderConfiguration` object using the `IdpLambdaArn` parameter.

You must create the Lambda function prior to creating your SMART on FHIR enabled data store. Once you create the data store, the Lambda ARN cannot be changed. To see the Lambda ARN you specified when the data store was created, use the `DescribeFHIRDatastore` API action.

For a FHIR REST request to succeed on a SMART on FHIR enabled data store, your Lambda function must do the following:

- Return a response in less than 1 second to the HealthLake data store endpoint.
- Decode the access token provided in the authorization header of the REST API request sent by the client application.
- Assign an IAM service role that has sufficient permissions to carry out the FHIR REST API request.
- The following claims are required to complete a FHIR REST API request. To learn more, see [Required claims](#).
 - `nbf`
 - `exp`
 - `isAuthorized`
 - `aud`
 - `scope`

When working with Lambda, you need to create an execution role and a resource-based policy in addition to your Lambda function. A Lambda's function's execution role is an IAM role that grants the function permission to access AWS services and resources needed at run time. The resource-based policy you provide must allow HealthLake to invoke your function on your behalf.

The sections in this topic describe an example request from a client application and decoded response, the steps needed to create an AWS Lambda function, and how to create a resource-based policy that HealthLake can assume.

- [Part 1: Creating a Lambda function](#)

- [Part 2: Creating a HealthLake service role used by the AWS Lambda function](#)
- [Part 3: Updating the Lambda function's execution role](#)
- [Part 4: Adding a resource policy to your Lambda function](#)
- [Part 5: Provisioning concurrency for your Lambda function](#)

Creating an AWS Lambda function

The Lambda function created in this topic is triggered when HealthLake receives a requests to a SMART on FHIR enabled data store. The request from the client application contains a REST API call, and an authorization header containing an access token.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/  
Authorization: Bearer i8hweunweunweofiwweoijewiwe
```

The example Lambda function in this topic uses AWS Secrets Manager to obscure credentials related to the authorization server. We strongly recommend not providing authorization server login details directly in a Lambda function.

Example validating a FHIR REST request containing an authorization bearer token

The example Lambda function shows you how to validate an FHIR REST request sent to a SMART on FHIR enabled data store. To see step-by-steps directions on how to implement this Lambda function, see [Creating a Lambda function using the AWS Management Console](#).

If the FHIR REST API request does not contain a valid data store endpoint, access token, and REST operation the Lambda function will fail. To learn more about the required authorization server elements, see [Required claims](#).

```
import base64  
import boto3  
import logging  
import json  
import os  
from urllib import request, parse  
  
logger = logging.getLogger()  
logger.setLevel(logging.INFO)  
  
## Uses Secrets manager to gain access to the access key ID and secret access key for  
the authorization server
```

```

client = boto3.client('secretsmanager', region_name="region-of-datastore")
response = client.get_secret_value(SecretId='name-specified-by-customer-in-secretsmanager')
secret = json.loads(response['SecretString'])
client_id = secret['client_id']
client_secret = secret['client_secret']

unencoded_auth = f'{client_id}:{client_secret}'
headers = {
    'Authorization': f'Basic {base64.b64encode(unencoded_auth.encode()).decode()}',
    'Content-Type': 'application/x-www-form-urlencoded'
}

auth_endpoint = os.environ['auth-server-base-url'] # Base URL of the Authorization
server
user_role_arn = os.environ['iam-role-arn'] # The IAM role client application will use
to complete the HTTP request on the datastore

def lambda_handler(event, context):
    if 'datastoreEndpoint' not in event or 'operationName' not in event or
    'bearerToken' not in event:
        return {}

    datastore_endpoint = event['datastoreEndpoint']
    operation_name = event['operationName']
    bearer_token = event['bearerToken']
    logger.info('Datastore Endpoint [{}], Operation Name:
    [{}]'.format(datastore_endpoint, operation_name))

    ## To validate the token
    auth_response = auth_with_provider(bearer_token)
    logger.info('Auth response: [{}]'.format(auth_response))
    auth_payload = json.loads(auth_response)
    ## Required parameters needed to be sent to the datastore endpoint for the HTTP
    request to go through
    auth_payload["isAuthorized"] = bool(auth_payload["active"])
    auth_payload["nbf"] = auth_payload["iat"]
    return {"authPayload": auth_payload, "iamRoleARN": user_role_arn}

## access the server
def auth_with_provider(token):
    data = {'token': token, 'token_type_hint': 'access_token'}

```



```
req = request.Request(url=auth_endpoint + '/v1/introspect',
data=parse.urlencode(data).encode(), headers=headers)
with request.urlopen(req) as resp:
    return resp.read().decode()
```

Creating a Lambda function using the AWS Management Console

The following procedure assumes you've already created the service role that you want HealthLake to assume when handling a FHIR REST API request on a SMART on FHIR enabled data store. If you have not created the service role, you can still create the Lambda function. You must add the ARN of service role before the Lambda function will work. To learn more about creating a service role and specifying it in the Lambda function, see [Creating a HealthLake service role for use in the AWS Lambda function used to decode a JWT](#)

To create a Lambda function (AWS Management Console)

1. Open the [Functions page](#) of the Lambda console.
2. Choose **Create function**.
3. Select **Author from scratch**.
4. Under **Basic information** enter a **Function name**. Under **Runtime** choose a python based runtime.
5. For **Execution role**, choose **Create a new role with basic Lambda permissions**.

Lambda creates an [execution role](#) that grants the function permission to upload logs to Amazon CloudWatch. The Lambda function assumes the execution role when you invoke your function, and uses the execution role to create credentials for the AWS SDK.

6. Choose the **Code** tab, and add the sample Lambda function.

If you've not yet created the service role for the Lambda function to use you'll need to create it before the sample Lambda function will work. To learn more about creating a service role for the Lambda function, see [Creating a HealthLake service role for use in the AWS Lambda function used to decode a JWT](#).

```
import base64
import boto3
import logging
import json
import os
from urllib import request, parse
```

```

logger = logging.getLogger()
logger.setLevel(logging.INFO)

## Uses Secrets manager to gain access to the access key ID and secret access key
for the authorization server
client = boto3.client('secretsmanager', region_name="region-of-datastore")
response = client.get_secret_value(SecretId='name-specified-by-customer-in-
secretsmanager')
secret = json.loads(response['SecretString'])
client_id = secret['client_id']
client_secret = secret['client_secret']

unencoded_auth = f'{client_id}:{client_secret}'
headers = {
    'Authorization': f'Basic {base64.b64encode(unencoded_auth.encode()).decode()}',
    'Content-Type': 'application/x-www-form-urlencoded'
}

auth_endpoint = os.environ['auth-server-base-url'] # Base URL of the Authorization
server
user_role_arn = os.environ['iam-role-arn'] # The IAM role client application will
use to complete the HTTP request on the datastore

def lambda_handler(event, context):
    if 'datastoreEndpoint' not in event or 'operationName' not in event or
    'bearerToken' not in event:
        return {}

    datastore_endpoint = event['datastoreEndpoint']
    operation_name = event['operationName']
    bearer_token = event['bearerToken']
    logger.info('Datastore Endpoint [{}], Operation Name:
    [{}]'.format(datastore_endpoint, operation_name))

    ## To validate the token
    auth_response = auth_with_provider(bearer_token)
    logger.info('Auth response: [{}]'.format(auth_response))
    auth_payload = json.loads(auth_response)
    ## Required parameters needed to be sent to the datastore endpoint for the HTTP
    request to go through
    auth_payload["isAuthorized"] = bool(auth_payload["active"])
    auth_payload["nbf"] = auth_payload["iat"]

```

```
    return {"authPayload": auth_payload, "iamRoleARN": user_role_arn}

## Access the server
def auth_with_provider(token):
    data = {'token': token, 'token_type_hint': 'access_token'}
    req = request.Request(url=auth_endpoint + '/v1/introspect',
    data=parse.urlencode(data).encode(), headers=headers)
    with request.urlopen(req) as resp:
        return resp.read().decode()
```

Modifying a Lambda function's execution role

After creating the Lambda function, you need to update the execution role to include the necessary permissions to call Secrets Manager. In Secrets Manager, each secret you create has an ARN. To apply the least privilege, the execution role should only have access to the resources needed for the Lambda function to execute.

You can modify a Lambda function's execution role by searching for it in the IAM console or by choosing **Configuration** in the Lambda console. To learn more about managing your Lambda functions execution role, see [Lambda execution role](#).

Example Lambda function execution role that grants access to GetSecretValue

Adding the IAM action `GetSecretValue` to execution role grants the necessary permission for the sample Lambda function to work.

At this point you've created a Lambda function that can be used to validate the access token provided as part of the FHIR REST request sent to your SMART on FHIR enabled data store.

Creating a HealthLake service role for use in the AWS Lambda function used to decode a JWT

Persona: IAM Administrator

A user who can add or remove IAM policies, and create new IAM identities.

Service role

A service role is an [IAM role](#) that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For

more information, see [Create a role to delegate permissions to an AWS service](#) in the *IAM User Guide*.

After the JSON Web Token (JWT) is decoded the authorization Lambda needs to also return an IAM role ARN. This role must have the necessary permissions to carry out the REST API request or it will fail due to insufficient permissions.

When setting up a custom policy using IAM it is best to grant the minimum permissions required. To learn more, see [Apply least-privilege permissions](#) in the *IAM User Guide*.

Creating a HealthLake service role to designate in the authorization Lambda function requires two steps.

- First, you need to create IAM policy. The policy must specify access to the FHIR resources that you have provided scopes for in the authorization server.
- Second, you need to create the service role. When you create the role you designate a trust relationship and attach the policy you created in step one. The trust relationship designates HealthLake as the service principal. You need to specify a HealthLake data store ARN and a AWS account ID in this step.

Creating a new IAM policy

The scopes you define in your authorization server determine what FHIR resources an authenticated user has access to in a HealthLake data store.

The IAM policy you create can be tailored to match the scopes you've defined.

The following actions in the `Action` element of an IAM policy statement can be defined. For each `Action` in the table you can define a `Resource` types. In HealthLake a data store is the only supported resource type that can be defined in the `Resource` element of an IAM permission policy statement.

Individual FHIR resources are not a resource that you can define as an element in a IAM permission policy.

Actions defined by HealthLake

Actions	Description	Access level	Resource type (Required)
CreateResource	Grants permission to create resource	Write	Datastore ARN: arn:aws:healthlake:your-region :111122223333 :datastore/fhir/your-datastore-id
DeleteResource	Grants permission to delete resource	Write	Datastore ARN: arn:aws:healthlake:your-region :111122223333 :datastore/fhir/your-datastore-id
ReadResource	Grants permission to read resource	Read	Datastore ARN: arn:aws:healthlake:your-region :111122223333 :datastore/fhir/your-datastore-id
SearchWithGet	Grants permission to search resources with GET method	Read	Datastore ARN: arn:aws:healthlake:your-region :111122223333 :datastore/fhir/your-datastore-id
SearchWithPost	Grants permission to search resources with POST method	Read	Datastore ARN: arn:aws:healthlake:your-region :111122223333 :datastore/fhir/your-datastore-id
StartFHIRExportJobWithPost	Grants permission to begin a FHIR Export job with GET	Write	Datastore ARN: arn:aws:healthlake:your-region :111122223333 :datastore/fhir/your-datastore-id
UpdateResource	Grants permission to update resource	Write	Datastore ARN: arn:aws:healthlake:your-region :111122223333 :datastore/fhir/your-datastore-id

To get started, you can use `AmazonHealthLakeFullAccess`. This policy would grant read, write, search, and export on all FHIR resources found in a data store. To grant read-only permissions on a data store use `AmazonHealthLakeReadOnlyAccess`.

To learn more about creating a custom policy using the AWS Management Console, AWS CLI, or IAM SDKs, see [Creating IAM](#) policies in the *IAM User Guide*.

Creating a service role for HealthLake (IAM console)

Use this procedure to create a service role. When you create a service you will also need to designate an IAM policy.

To create the service role for HealthLake (IAM console)

1. Sign in to the AWS Management Console and open the IAM console at <https://console.aws.amazon.com/iam/>.
2. In the navigation pane of the IAM console, choose **Roles**.
3. Then, choose **Create role**.
4. On the **Select trust entity** page, choose **Custom trust policy**.
5. Next, under **Custom trust policy** update the sample policy as follows. Replace **your-account-id** with your account number, and add the ARN of the data store you want to use in your import or export jobs.
6. Then, choose **Next**.
7. On the **Add permissions** page, choose the policy that you want the HealthLake service to assume. To find your policy, search for it under **Permissions policies**.
8. Then, choose **Attach policy**.
9. Then on the **Name, review, and create** page under **Role name** enter a name.
10. (Optional) Then under **Description**, add a short description for your role.
11. If possible, enter a role name or role name suffix to help you identify the purpose of this role. Role names must be unique within your AWS account. They are not distinguished by case. For example, you cannot create roles named both **PRODRROLE** and **prodrole**. Because various entities might reference the role, you cannot edit the name of the role after it has been created.
12. Review the role details, and then choose **Create role**.

To learn how to specify the role ARN in the sample Lambda function, see [Creating an AWS Lambda function](#).

Lambda execution role

A Lambda function's execution role is an IAM role that grants the function permission to access AWS services and resources. This page provides information on how to create, view, and manage a Lambda function's execution role.

By default, Lambda creates an execution role with minimal permissions when you create a new Lambda function using the AWS Management Console. To manage the permissions granted in the execution role, see [Creating an execution role in the IAM console](#) in the *Lambda Developer Guide*.

The sample Lambda function provided in this topic uses Secrets Manager to obscure the authorization server's credentials.

As with any IAM role you create it is important to follow the least privilege best practice. During the development phase, you might sometimes grant permissions beyond what is required. Before publishing your function in the production environment, as a best practice, adjust the policy to include only the required permissions. For more information, see [Apply least-privilege](#) in the *IAM User Guide*.

Allow HealthLake to trigger your Lambda function

So HealthLake can invoke the Lambda function on your behalf, you must do following:

- You need to set `IdpLambdaArn` equal to the ARN of the Lambda function you want HealthLake to invoke in the `CreateFHIRDatastore` request.
- You need a resource-based policy allowing HealthLake to invoke the Lambda function on your behalf.

When HealthLake receives a FHIR REST API request on a SMART on FHIR enabled data store, it needs permissions to invoke the Lambda function specified at data store creation on your behalf. To grant HealthLake access, you'll use a resource-based policy. To learn more about creating a resource-based policy for a Lambda function, see [Allowing an AWS service to call a Lambda function](#) in the *AWS Lambda Developer Guide*.

Provisioning concurrency for your Lambda function

Important

HealthLake requires that the maximum run time for your Lambda function be less than one second (1000 milliseconds).

If your Lambda function exceeds the run time limit you get a `Timeout` exception.

To avoid getting this exception, we recommend configuring provisioned concurrency. By allocating provisioned concurrency before an increase in invocations, you can ensure that all requests are

served by initialized instances with low latency. To learn more about configuring provisioned concurrency, see [Configuring provisioned concurrency](#) in the *Lambda Developer Guide*

To see the average run time for your Lambda function currently use the **Monitoring** page for your Lambda function on the Lambda console. By default, the Lambda console provides a **Duration** graph which shows you the average, minimum, and maximum amount of time your function code spends processing an event. To learn more about monitoring Lambda functions, see [Monitoring functions in the Lambda console](#) in the *Lambda Developer Guide*.

If you have already provisioned concurrency for your Lambda function and want to monitor it, see [Monitoring concurrency](#) in the *Lambda Developer Guide*.

Using fine-grained authorization with a SMART on FHIR enabled HealthLake data store

[Scopes](#) alone do not provide you with the necessary specificity about what data a requester is authorized to access in a data store. Using fine-grained authorization enables a higher level of specificity when granting access to a SMART on FHIR enabled HealthLake data store. To use fine-grained authorization, set `FineGrainedAuthorizationEnabled` equal to `True` in the `IdentityProviderConfiguration` parameter of your `CreateFHIRDatastore` request.

If you enabled fine-grained authorization, your authorization server returns a `fhirUser` scope in the `id_token` along with the access token. This permits information about the User to be retrieved by client application. The client application should treat the `fhirUser` claim as the URI of a FHIR resource representing the current user. This can be `Patient`, `Practitioner`, or `RelatedPerson`. The authorization server's response also includes a `user/` scope that defines what data the user can access. This uses the syntax defined for scopes related to FHIR resource specific scopes:

```
user/(fhir-resource | '*').('read' | 'write' | '*')
```

The following are examples of how fine-grained authorization can be used to further specify data access related FHIR resource types.

- When `fhirUser` is a `Practitioner`, fine-grained authorization determines the collection of patients that the user can access. Access to `fhirUser` is allowed for only those patients where the `Patient` has reference to the `fhirUser` as a General Practitioner.

```
Patient.generalPractitioner : [{Reference(Practitioner)}]
```


- When `fhirUser` is a `Patient` or `RelatedPerson` and the patient referenced in the request is different from the `fhirUser`, fine-grained authorization determines access to `fhirUser` for the requested patient. Access is allowed when there is a relationship specified in requested `Patient` resource.

```
Patient.link.other : {Reference(Patient|RelatedPerson)}
```

Fetching the SMART on FHIR Discovery Document

SMART defines a Discovery Document that allows clients to learn the authorization endpoint URLs and features a HealthLake data store supports. This information helps clients direct authorization requests to the right endpoint and construct authorization requests the HealthLake data store supports.

For a client application to make a successful FHIR REST request to HealthLake, it must gather the authorization requirements defined by the HealthLake data store. A bearer token (authorization) is *not* required for this request to succeed..

To request the Discovery Document for a HealthLake data store

1. Collect HealthLake region and `datastoreId` values. For more information, see [Getting data store properties](#).
2. Construct a URL for the request using the collected values for HealthLake region and `datastoreId`. Append `/.well-known/smart-configuration` to the endpoint of the URL. To view the entire URL path in the following example, scroll over the **Copy** button.

```
https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/.well-known/smart-configuration
```

3. Send the request using GET with [AWS Signature Version 4](#) signing protocol. To view the entire example, scroll over the **Copy** button.

curl

```
curl --request GET \  
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/.well-known/  
smart-configuration \  
  --aws-sigv4 'aws:amz:region:healthlake' \  

```

```
--user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \  
--header "x-amz-security-token:$AWS_SESSION_TOKEN" \  
--header 'Accept: application/json'
```

The Discovery Document for the HealthLake data store returns as a JSON blob, where you can find the `authorization_endpoint` and the `token_endpoint`, along with the specifications and defined capabilities for the data store.

```
{  
  "authorization_endpoint": "https://oidc.example.com/authorize",  
  "token_endpoint": "https://oidc.example.com/oauth/token",  
  "capabilities": [  
    "launch-ehr",  
    "client-public"  
  ]  
}
```

Both the `authorization_endpoint` and the `token_endpoint` are required to launch a client application.

- **Authorization endpoint** — The URL needed to authorize a client application or user.
- **Token endpoint** — The endpoint of the authorization server the client application uses to communicate with.

Making a FHIR REST API request on a SMART-enabled HealthLake data store

You can make FHIR REST API requests on a SMART on FHIR-enabled HealthLake data store. The following example shows a request from client application containing a JWT in the authorization header and how Lambda should decode the response. After the client application request is authorized and authenticated, it must receive a bearer token from the authorization server. Use the bearer token in the authorization header when sending a FHIR REST API request on a SMART on FHIR-enabled HealthLake data store.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/[ID]  
Authorization: Bearer auth-server-provided-bearer-token
```

Because a bearer token was found in the authorization header and no AWS IAM identity was detected HealthLake invokes the Lambda function specified when the SMART on FHIR enabled HealthLake data store was created. When the token is successfully decoded by your Lambda function, the following example response is sent to HealthLake.

```
{
  "authPayload": {
    "iss": "https://authorization-server-endpoint/oauth2/token", # The issuer
    identifier of the authorization server
    "aud": "https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/", #
    Required, data store endpoint
    "iat": 1677115637, # Identifies the time at which the token was issued
    "nbf": 1677115637, # Required, the earliest time the JWT would be valid
    "exp": 1997877061, # Required, the time at which the JWT is no longer valid
    "isAuthorized": "true", # Required, boolean indicating the request has been
    authorized
    "uid": "100101", # Unique identifier returned by the auth server
    "scope": "system/*.*" # Required, the scope of the request
  },
  "iamRoleARN": "iam-role-arn" #Required, IAM role to complete the request
}
```

FHIR R4 support for AWS HealthLake

AWS HealthLake supports the FHIR R4 specification for health data exchange. The following sections provide supporting information on how HealthLake utilizes the FHIR R4 specification to help you [manage](#) and [search](#) FHIR resources in your HealthLake data store using FHIR R4 RESTful APIs.

Topics

- [FHIR R4 Capability Statement for AWS HealthLake](#)
- [FHIR profile validations for HealthLake](#)
- [FHIR R4 supported resource types for HealthLake](#)
- [FHIR R4 search parameters for HealthLake](#)
- [FHIR R4 \\$operations for HealthLake](#)

FHIR R4 Capability Statement for AWS HealthLake

To find the FHIR-related capabilities (behaviors) of an active HealthLake data store, you must retrieve its Capability Statement. The Capability Statement is used as a statement of actual server functionality or a statement of required or desired server implementation. The FHIR [capabilities](#) interaction retrieves information about HealthLake data store capabilities and which portions of the FHIR specification it supports. HealthLake validates FHIR resource types according to the FHIR R4 [StructureDefinition](#) resource.

To get the Capability Statement for a HealthLake data store

1. Collect HealthLake region and datastoreId values. For more information, see [Getting data store properties](#).
2. Construct a URL for the request using the collected values for HealthLake region and datastoreId. Also include the FHIR metadata element in the URL. To view the entire URL path in the following example, scroll over the **Copy** button.

```
https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/metadata
```

3. Send the request. The FHIR capabilities interaction uses a GET request with [AWS Signature Version 4](#) signing protocol. The following curl example gets the Capability Statement for the HealthLake data store specified by the datastoreId. To view the entire example, scroll over the **Copy** button.

curl

```
curl --request GET \  
  'https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/metadata \  
  --aws-sigv4 'aws:amz:region:healthlake' \  
  --user "$AWS_ACCESS_KEY_ID:$AWS_SECRET_ACCESS_KEY" \  
  --header "x-amz-security-token:$AWS_SESSION_TOKEN" \  
  --header 'Accept: application/json'
```

You will receive a 200 HTTP response code and the Capability Statement for your HealthLake data store. For more information, see [CapabilityStatement](#) in the FHIR R4 documentation.

FHIR profile validations for HealthLake

AWS HealthLake supports the base [FHIR R4 specification](#). Included in the base FHIR R4 specification are FHIR Profiles. Profiles are used on a FHIR resource type to define a more specific resource type definition using constraints and/or extensions on the base resource type. For example, a FHIR Profile can identify mandatory fields such as extensions and value sets. A resource can support multiple profiles. All HealthLake data stores support using FHIR Profiles.

Note

Adding a FHIR profile is *not* required when adding data to a HealthLake data store. If a FHIR profile is not specified when a resource is added or updated, the resource is validated against only the base FHIR R4 schema.

FHIR profiles, to which FHIR resources conform to, are included in resources before they are imported into HealthLake. Therefore, FHIR profiles are validated by HealthLake during import.

FHIR Profiles are specified in an implementation guide. A FHIR Implementation Guide (IG) is a set of instructions that describe how to use the FHIR standard for a specific purpose. HealthLake validates FHIR Profiles defined in the following implementation guides.

FHIR profiles supported by AWS HealthLake

Name	Version	Implementation guide	Capability
US Core	3.1.1	http://hl7.org/fhir/us/core/STU3.1.1/	Default
US Core	4.0.0	https://hl7.org/fhir/us/core/STU4/index.html	Supported
US Core	5.0.1	https://hl7.org/fhir/us/core/STU5.0.1/index.html	Supported
US Core	6.1.0	https://hl7.org/fhir/us/core/STU6.1/index.html	Supported
US Core	7.0.0	https://hl7.org/fhir/us/core/STU7/	Supported

Name	Version	Implementation guide	Capability
UK Core	2.0.1	https://simplifier.net/guide/uk-core-implementation-guide-stu2/Home/ProfilesandExtensions/ProfilesIndex?version=2.0.1	Supported
CARIN Blue Button	1.1.0	http://hl7.org/fhir/us/carin-bb/STU1.1/	Default
CARIN Blue Button	1.0.0	https://hl7.org/fhir/us/carin-bb/STU1/	Supported
Da Vinci Payer Data Exchange	1.0.0	https://hl7.org/fhir/us/davinci-pdex/	Default
Da Vinci Health Record Exchange (HREx)	0.2.0	https://hl7.org/fhir/us/davinci-hrex/2020Sep/	Default
DaVinci PDEX Plan Net	1.1.0	https://hl7.org/fhir/us/davinci-pdex-plan-net/STU1.1/	Default
DaVinci PDEX Plan Net	1.0.0	https://hl7.org/fhir/us/davinci-pdex-plan-net/STU1/	Supported
DaVinci Payer Data Exchange (PDex) US Drug Formulary	1.1.0	https://hl7.org/fhir/us/davinci-drug-formulary/STU1.1/	Default
DaVinci Payer Data Exchange (PDex) US Drug Formulary	1.0.1	https://hl7.org/fhir/us/davinci-drug-formulary/STU1.0.1/	Supported
NCQA HEDIS® Implementation Guide	0.3.1	https://www.ncqa.org/resources/hedis-ig-resource-page/	Default

Name	Version	Implementation guide	Capability
International Patient Summary (IPS)	2.0.0 bal lot	https://hl7.org/fhir/uv/ips/2024Sep/	Default
Quality Measure	5.0.0	https://registry.fhir.org/package/hl7.fhir.us.cqfmeasures%7C5.0.0	Default
Genomics Reporting	3.0.0	https://build.fhir.org/ig/HL7/genomics-reporting/index.html	Default
National Health Authority's Ayushman Bharat Digital Mission (ABDM)	2.0	https://www.nrces.in/ndhm/fhir/r4/index.html	Default

Validating FHIR profiles specified in a resource

For a FHIR Profile to be validated add it to the `profile` element of individual resources using the profile URL designated in the implementation guide.

FHIR Profiles are validated when you add a new resource to your data store. To add a new resource, you can use the `StartFHIRImportJob` API operation, make a POST request to add a new resource, or make `PUT` to update an existing resource.

Example – To see which FHIR profile is referenced in a resource

The profile URL is added to the `profile` element in the `"meta" : "profile"` key-value pair. This resource was truncated for clarity.

```
{
  "resourceType": "Patient",
  "id": "abcd1234efgh5678hijk9012",
  "meta": {
    "lastUpdated": "2023-05-30T00:48:07.8443764-07:00",
    "profile": [
      "http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient"
    ]
  }
}
```

```
}  
}
```

Example – How to reference a non-default supported FHIR profile

To validate against a supported non-default profile (e.g. CarinBB 1.0.0) - add the profile URL with version (separated by '|') and the base profile URL in the `meta.profile` element. This example resource was truncated for clarity.

```
{  
  "resourceType": "ExplanationOfBenefit",  
  "id": "sample-EOB",  
  "meta": {  
    "lastUpdated": "2024-02-02T05:56:09.4+00:00",  
    "profile": [  
      "http://hl7.org/fhir/us/carin-bb/StructureDefinition/C4BB-  
ExplanationOfBenefit-Pharmacy|1.0.0",  
      "http://hl7.org/fhir/us/carin-bb/StructureDefinition/C4BB-ExplanationOfBenefit-  
Pharmacy"  
    ]  
  }  
}
```

FHIR R4 supported resource types for HealthLake

The following table lists the FHIR R4 resource types supported by AWS HealthLake. For more information, see [Resource Index](#) in the FHIR R4 documentation.

FHIR R4 resource types supported by HealthLake

Account	DetectedIssue	Invoice	Practitioner
ActivityDefinition	Device	Library	PractitionerRole
AdverseEvent	DeviceDefinition	Linkage	Procedure
AllergyIntolerance	DeviceMetric	List	Provenance
Appointment	DeviceUseStatement	Location	Questionnaire
AppointmentResponse	DeviceRequest	Measure	QuestionnaireResponse

AuditEvent - <i>See Note</i>	DiagnosticReport	MeasureReport	RelatedPerson
Binary	DocumentManifest	Media	RequestGroup
BodyStructure	DocumentReference	Medication	ResearchStudy
Bundle - <i>See Note</i>	EffectEvidenceSynt hesis	MedicationAdminist ration	ResearchSubject
CapabilityStatement	Encounter	MedicationDispense	RiskAssessment
CarePlan	Endpoint	MedicationKnowledg e	RiskEvidenceSynthe sis
CareTeam	EpisodeOfCare	MedicationRequest	Schedule
ChargeItem	EnrollmentRequest	MedicationStatement	ServiceRequest
ChargeItemDefinition	EnrollmentResponse	MessageHeader	Slot
Claim	ExplanationOfBenefit	MolecularSequence	Specimen
ClaimResponse	FamilyMemberHistor y	NutritionOrder	StructureDefinition
Communication	Flag	Observation	StructureMap
Communica tionRequest	Goal	OperationOutcome - <i>See Note</i>	Substance
Composition	Group	Organization	SupplyDelivery
ConceptMap	GuidanceResponse	OrganizationAffili ation	SupplyRequest
Condition	HealthcareService	Parameters - <i>See Note</i>	Task
Consent	ImagingStudy	Patient	ValueSet
Contract	Immunization	PaymentNotice	VisionPrescription

Coverage	ImmunizationEvaluation	PaymentReconciliation	VerificationResult - <i>See Note</i>
CoverageEligibilityRequest	ImmunizationRecommendation	Person	
CoverageEligibilityResponse	InsurancePlan	PlanDefinition	

FHIR specifications and HealthLake

- You cannot make GET or POST requests with FHIR `OperationOutcome` and `Parameters` resource types.
- **AuditEvent** — An `AuditEvent` resource can be created or read, but it cannot be updated or deleted.
- **Bundle** — There are multiple ways HealthLake manages `Bundle` requests. For more details, see [Bundling FHIR resources](#).
- **VerificationResult** — This resource type is only supported for data stores created after December 09, 2023.

FHIR R4 search parameters for HealthLake

Use FHIR [search](#) interaction to search a set of FHIR resources in a HealthLake data store based on some filter criteria. The search interaction can be performed using either a GET or POST request. For searches that involve personally identifiable information (PII) or protected health information (PHI), it's recommended to use POST requests, as PII and PHI is added as part of the request body and is encrypted in transit.

Note

The FHIR search interaction described in this chapter is built in conformance to the HL7 FHIR R4 standard for health care data exchange. Because it is a representation of a HL7 FHIR service, it is not offered through AWS CLI and AWS SDKs. For more information, see [search](#) in the FHIR R4 RESTful API documentation.

You can also query HealthLake data stores with SQL using Amazon Athena. For more information, see [Integrating](#).

HealthLake supports the following subset of FHIR R4 search parameters. For more information, see [FHIR R4 search parameters for HealthLake](#).

Supported search parameter types

The following table shows the supported search parameter types in HealthLake.

Supported search parameters types

Search parameter	Description
_id	Resource id (not a full URL)
_lastUpdated	Date last updated. Server has discretion on the boundary precision.
_tag	Search by a resource tag.
_profile	Search for all resources tagged with a profile.
_security	Search on security labels applied to this resource.
_source	Search on where the resource comes from.
_text	Search on the narrative of the resource.
createdAt	Search on custom extension createdAt.

Note

The following search parameters are only supported for datastores created after December 09, 2023 : _security, _source, _text, createdAt.

The following table shows examples of how to modify query strings based on specified data types for a given resource type. For clarity, special characters in the examples column have not been encoded. To make a successful query, ensure that the query string has been properly encoded.

Search parameter examples

Search Parameter Types	Details	Examples
Number	Searches for a numerical value in a specified resource. Significant figures are observed. The number of significant digits are specific in by search parameter value, excluding leading zeros. Comparison prefixes are allowed.	<code>[parameter]=100</code> <code>[parameter]=1e2</code> <code>[parameter]=1t100</code>
Date/DateTime	Searches for a specific date or time. The expected format is <code>yyyy-mm-ddThh:mm:ss[Z (+ -)hh:mm]</code> but can vary. Accepts the following data types: date, dateTime, instant, Period, and Timing. For more details using these data types in searches, see date in the FHIR R4 RESTful API documentation. Comparison prefixes are allowed.	<code>[parameter]=eq2013-01-14</code> <code>[parameter]=gt2013-01-14T10:00</code> <code>[parameter]=ne2013-01-14</code>
String	Searches for a sequence of characters in a case-sensitive manner.	<code>[base]/Patient?given=eve</code>

Search Parameter Types	Details	Examples
	Supports both <code>HumanName</code> and <code>Address</code> types. For more details, see the HumanName data type entry and the Address data type entries in the FHIR R4 documentation. Advanced search is supported using <code>:text</code> modifiers.	<code>[base]/Patient?given:contains=eve</code>
Token	Searches for a close-to-exact match against a string of characters, often compared to a pair of medical code values. Case sensitivity is linked to the code system used when creating a query. Subsumption-based queries can help reduce issues linked to case sensitivity. For clarity the <code> </code> has not been encoded.	<code>[parameter]=[system] [code]</code> : Here <code>[system]</code> refers a coding system, and <code>[code]</code> refers to code value found within that specific system. <code>[parameter]=[code]</code> : Here your input will match either a code or a system. <code>[parameter]= [code]</code> : Here your input will match a code, and the system property has no identifier.

Search Parameter Types	Details	Examples
Composite	Searches for multiple parameters within a single resource type, using the modifiers\$ and , operation. Comparison prefixes are allowed.	/Patient?language=FR,NL&language=EN Observation?component-code-value-quantity=http://loinc.org 8480-6\$lt60 [base]/Group?characteristic-value=gender\$mixed
Quantity	Searches for a number, system, and code as values. A number is required, but system and code are optional. Based on the Quantity data type. For more details, see Quantity in the FHIR R4 documentation. Uses the following assumed syntax [parameter]=[prefix][number] [system] [code]	[base]/Observation?value-quantity=5.4 http://unitsofmeasure.org mg [base]/Observation?value-quantity=5.4 http://unitsofmeasure.org mg [base]/Observation?value-quantity=5.4 http://unitsofmeasure.org mg [base]/Observation?value-quantity=le5.4 http://unitsofmeasure.org mg
Reference	Searches for references to other resources.	[base]/Observation?subject=Patient/23 test

Search Parameter Types	Details	Examples
URI	Searches for a string of characters that unambiguously identifies a particular resource.	[base]/ValueSet?url=http://acme.org/fhir/ValueSet/123
Special	Searches based on integrated medical NLP extensions.	

Advanced search parameters supported by HealthLake

HealthLake supports the following advanced search parameters.

Name	Description	Example	Capability
<code>_include</code>	Used to request that additional resources be returned in a search request. It returns resources which are referenced by the target resource instance.	Encounter? _include=Encounter:subject	
<code>_revinclude</code>	Used to request that additional resources be returned in a search request. It returns resources that reference the primary resource instance.	Patient?_id= patient-identifier &_revinclude=Encounter:patient	
<code>_summary</code>	Summary can be used to request a subset of the resource.	Patient?_summary=text	The following summary parameters are supported: :_summary=true , :_summary=false , :_summary=text , :_summary=data .

Name	Description	Example	Capability
<code>_elements</code>	Request a specific set of elements to be returned as part of a resource in the search results.	<code>Patient?_elements=identifier,active,link</code>	
<code>_total</code>	Returns the number of resources that match the search parameters.	<code>Patient?_total=accurate</code>	Support <code>_total=accurate</code> , <code>_total=none</code> .
<code>_sort</code>	Indicate the sort order of the returned search results using a comma-separated list. The <code>-</code> prefix can be used for any sort rule in the comma-separated list to indicate descending order.	<code>Observation?_sort=status,-date</code>	Support sort by fields with types Number, String, Quantity, Token, URI, Reference . Sort by Date is only supported for data stores created after December 09, 2023. Support up to 5 sort rules.
<code>_count</code>	Control how many resources are returned per page of the search bundle.	<code>Patient?_count=100</code>	Maximum page size is 100.
<code>chaining</code>	Search elements of referenced resources. The <code>.</code> directs the chained search to the element within the referenced resource.	<code>DiagnosticReport?subject:Patient.name=peter</code>	
<code>reverse chaining (_has)</code>	Search for a resource based on the elements of resources that refer to them.	<code>Patient?_has:Observation:patient:code=1234-5</code>	

_include

Using `_include` in a search query allows for additional specified FHIR resources to also be returned. Use `_include` to include resources that are linked forward.

Example – To use `_include` to find the patients or the group of patients who have been diagnosed with a cough

You would search on the `Condition` resource type specifying the diagnostic code for cough, and then using `_include` specify that you want the subject of that diagnosis returned too. In the `Condition` resource type subject refers to either the patient resource type or the group resource type.

For clarity, special characters in the example have not been encoded. To make a successful query ensure that the query string has been properly encoded.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/  
Condition?code=49727002&_include=Condition:subject
```

_revinclude

Using `_revinclude` in a search query allows for additional specified FHIR resources to also be returned. Use `_revinclude` to include resources that are linked backwards.

Example – To use `_revinclude` to include related `Encounter` and `Observation` resource types linked to a specific Patient

To make this search, you would first define the individual Patient by specifying their identifier in the `_id` search parameter. Then you would specify additional FHIR resources using the structure `Encounter:patient` and `Observation:patient`.

For clarity, special characters in the example have not been encoded. To make a successful query ensure that the query string has been properly encoded.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/  
Patient?_id=patient-  
identifier&_revinclude=Encounter:patient&_revinclude=Observation:patient
```

_summary

Using `_summary` in a search query allows user to request a subset of the FHIR resource. It can contain one of the following values: `true`, `text`, `data`, `false`. Any other values will be

treated as invalid. The returned resources will be marked with ' SUBSETTED ' in meta.tag, to indicate that resources are incomplete.

- `true`: Return all supported elements that are marked as 'summary' in the base definition of the resource(s).
- `text`: Return only the 'text', 'id', 'meta' elements, and only top-level mandatory elements.
- `data`: Return all parts except the 'text' element.
- `false`: Return all parts of the resource(s)

In a single search request, `_summary=text` cannot be combined with `_include` or `_revinclude` search parameters.

Example – Get “text” element of Patient resources in a data store.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?
_summary=text
```

`_elements`

Using `_elements` in a search query allows for specific FHIR resource elements to be requested. The returned resources will be marked with ' SUBSETTED ' in meta.tag, to indicate that resources are incomplete.

The `_elements` parameter consists of a comma-separated list of base element names such as elements defined at the root level in the resource. Only elements that are listed are to be returned. If `_elements` parameter values contain invalid elements, server will ignore them and return mandatory elements and valid elements.

`_elements` will not be applicable to included resources(returned resources whose search mode is `include`).

In a single search request, `_elements` cannot be combined with `_summary` search parameters.

Example – Get “identifier”, “active”, “link” elements of Patient resources in your HealthLake data store.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?
_elements=identifier,active,link
```

_total

Using `_total` in a search query will return number of resources that match the requested search parameters. HealthLake will return the total number of matched resources(returned resources whose search mode is `match`) in the `Bundle.total` of search response.

`_total` supports the `accurate`, `none` parameter values. `_total=estimate` is not supported. Any other values will be treated as invalid. `_total` is not applicable to the included resources(returned resources whose search mode is `include`).

Example – Get the total number of Patient resources in a data store:

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?
_total=accurate
```

_sort

Using `_sort` in the search query arranges the results in a specific order. The results are ordered based on the comma-separated list of sort rules in priority order. The sort rules should be valid search parameters. Any other values will be treated as invalid.

In a single search request, you can use up to 5 sort search parameters. You can optionally use a `-` prefix to indicate descending order. Server will sort on ascending order by default.

The supported sort search parameter types are: `Number`, `String`, `Date`, `Quantity`, `Token`, `URI`, `Reference`. If a search parameter refers to an element that is nested, this search parameter is not supported for sort. For example, search on 'name' of resource type `Patient` refers to `Patient.name` element with `HumanName` data type is considered as nested. Thus, sort on `Patient` resources by 'name' is not supported.

Example – Get Patient resources in a data store and sort them by birthdate in ascending order:

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?
_sort=birthdate
```

_count

The parameter `_count` is defined as an instruction to the server regarding how many resources should be returned in a single page.

The maximum page size is 100. Any values greater than 100 is invalid. `_count=0` is not supported.

Example – Search for the Patient resource and set search page size to 25:

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?_count=25
```

Chaining and Reverse Chaining(_has)

Chaining and reverse chaining in FHIR provide a more efficient and compact way to obtain interconnected data, reducing the need for multiple separate queries and making data retrieval more convenient for developers and users.

If any level of recursion return more than 100 results, HealthLake will return 4xx to protect data store from being overloaded and causing multiple paginations.

Example – Chaining - Gets all DiagnosticReport which refer to a Patient where Patient name is peter.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/DiagnosticReport?subject:Patient.name=peter
```

Example – Reverse Chaining - Get Patient resources, where the patient resource is referred to by at least one Observation where the observation has a code of 1234, and where the Observation refers to the patient resource in the patient search parameter.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient?_has:Observation:patient:code=1234
```

Supported search modifiers

Search modifiers are used with string-based fields. All search modifiers in HealthLake use Boolean-based logic. For example, you could specify `:contains` to specify that larger string field should include a small string in order for it to be included in your search results.

Supported search modifiers

Search modifier	Type
<code>:missing</code>	All parameters except Composite
<code>:exact</code>	String

Search modifier	Type
:contains	String
:not	Token
:text	Token
:identifier	Reference
:below	URI

Supported search comparators

You can use search comparators to control the nature of the matching in a search. You can use comparators when searching on number, date, and quantity fields. The following table lists search comparators and their definitions that are supported by HealthLake.

Supported search comparators

Search comparator	Description
eq	The value for the parameter in the resource is equal to the provided value.
ne	The value for the parameter in the resource is not equal to the provided value.
gt	The value for the parameter in the resource is greater than the provided value.
lt	The value for the parameter in the resource is less than the provided value.
ge	The value for the parameter in the resource is greater or equal to the provided value.
le	The value for the parameter in the resource is less or equal to the provided value.

Search comparator	Description
sa	The value for the parameter in the resource starts after the provided value.
eb	The value for the parameter in the resource ends before the provided value.

FHIR search parameters not supported by HealthLake

HealthLake supports all FHIR search parameters with the exception of those listed in the following table. For a full list of FHIR search parameters, see the [FHIR search parameter registry](#).

Unsupported search parameters

Bundle-composition	Location-near
Bundle-identifier	Consent-source-reference
Bundle-message	Contract-patient
Bundle-type	Resource-content
Bundle-timestamp	Resource-query

FHIR R4 \$operations for HealthLake

FHIR \$ operations (also called "dollar operations") are special server-side functions that extend beyond standard CRUD (Create, Read, Update, Delete) operations in the FHIR specification. These operations are identified by the "\$" prefix and enable complex processing, data transformation, and bulk operations that would be difficult or inefficient to perform using standard REST API calls. \$ Operations can be invoked at the system level, resource type level, or on specific resource instances, providing flexible ways to interact with FHIR data. AWS HealthLake supports multiple FHIR \$operations. Please refer to each individual pages below for additional details.

Topics

- [Deleting Resource Types with \\$bulk-delete](#)
- [Generating Clinical Documents with \\$document](#)

- [Permanently Removing Resources with \\$erase](#)
- [Getting patient data with Patient/\\$everything](#)
- [Retrieving ValueSet Codes with \\$expand](#)
- [Exporting HealthLake data with FHIR \\$export](#)
- [Retrieving Concept Details with \\$lookup](#)
- [Removing Patient Compartment Resources with \\$purge](#)
- [Validating FHIR Resources with \\$validate](#)

Deleting Resource Types with \$bulk-delete

AWS HealthLake supports the \$bulk-delete operation, enabling deletion of all resources of a specific type within a datastore. This operation is particularly useful when you need to:

- Perform seasonal auditing and clean-up
- Manage data lifecycle at scale
- Remove specific resource types
- Comply with data retention policies

Usage

The \$bulk-delete operation can be invoked using POST methods:

```
POST [base]/[ResourceType]/$bulk-delete?  
hardDelete=false&deleteVersionHistory=false&deleteAuditEvent=true
```

Parameters

Parameter	Type	Required	Default	Description
isHardDelete	boolean	No	false	When true, permanently removes resources from storage
deleteVersionHistory	boolean	No	false if isHardDelete is false,	When true, deletes all historical versions, must be used with hardDelete = true or ignored

Parameter	Type	Required	Default	Description
			true if isHardDelete is true	
deleteAuditEvent	boolean	No	true	When true, deletes associated audit events
_since	string	No	Datastore creation time	When entered, selects the starting cutoff time to find resources based on their lastModified time. Cannot be used with start or end
start	string	No	Datastore creation time	When entered, selects the cutoff time to find resources based on their lastModified time. Can be used with end
end	string	No	Job submission time	When entered, selects the ending cutoff time to find resources based on their lastModified time

Examples

Example Request

```
POST [base]/Observation/$bulk-delete?hardDelete=true
```

Example Response

```
{
  "resourceType": "OperationOutcome",
  "id": "bulk-delete-job",
  "issue": [
    {
      "severity": "information",
      "code": "informational",
```



```
    "diagnostics": "Bulk delete job started successfully. Job ID:
12345678-1234-1234-1234-123456789012"
  }
]
}
```

Job Status

To check the status of a bulk delete job:

```
GET [base]/$bulk-delete/[jobId]
```

The operation returns job status information:

```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "jobId",
      "valueString": "12345678-1234-1234-1234-123456789012"
    },
    {
      "name": "jobStatus",
      "valueString": "COMPLETED"
    },
    {
      "name": "totalResourcesDeleted",
      "valueInteger": 1256
    },
    {
      "name": "startTime",
      "valueInstant": "2023-06-15T10:00:00Z"
    },
    {
      "name": "endTime",
      "valueInstant": "2023-06-15T10:05:23Z"
    }
  ]
}
```

Behavior

The \$bulk-delete operation:

1. Processes asynchronously to handle large volumes of resources
2. Maintains ACID transactions for data integrity
3. Provides job status tracking with resource deletion counts
4. Supports both soft and hard deletion modes
5. Includes comprehensive audit logging of deletion activities
6. Allows for selective deletion of historical versions and audit events

Audit Logging

The `$bulk-delete` operation logs as `StartFHIRBulkDeleteJob` and `DescribeFHIRBulkDeleteJob` with detailed operation information.

Limitations

- Hard-deleted resources will not appear in search responses
- Resources being deleted through this operation may be temporarily inaccessible during processing
- Storage metering is adjusted on historical versions only - `deleteVersionHistory=false` will not adjust datastore storage

Generating Clinical Documents with `$document`

AWS HealthLake now supports the `$document` operation for Composition resources, enabling you to generate a complete clinical document by bundling the Composition with all its referenced resources into a single cohesive package. This operation is essential for healthcare applications that need to:

- Create standardized clinical documents
- Exchange complete patient records
- Store comprehensive clinical documentation
- Generate reports that include all relevant context

Usage

The `$document` operation can be invoked on Composition resources using both GET and POST methods:

Supported Operations

```
GET/POST [base]/Composition/[id]/$document
```

Supported Parameters

HealthLake supports the following FHIR \$document parameter:

Parameter	Type	Required	Default	Description
<code>persist</code>	boolean	No	false	Boolean indicating whether the server should store the generated document bundle

Examples

GET Request

```
GET [base]/Composition/180f219f-97a8-486d-99d9-ed631fe4fc57/$document?persist=true
```

POST Request with Parameters

```
POST [base]/Composition/180f219f-97a8-486d-99d9-ed631fe4fc57/$document
Content-Type: application/fhir+json
```

```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "persist",
      "valueBoolean": true
    }
  ]
}
```

Sample Response

The operation returns a Bundle resource of type "document" containing the Composition and all referenced resources:

```
{
  "resourceType": "Bundle",
  "id": "180f219f-97a8-486d-99d9-ed631fe4fc57",
  "type": "document",
  "identifier": {
    "system": "urn:ietf:rhc:3986",
    "value": "urn:uuid:0c3151bd-1cbf-4d64-b04d-cd9187a4c6e0"
  },
  "timestamp": "2024-06-21T15:30:00Z",
  "entry": [
    {
      "fullUrl": "http://example.org/fhir/Composition/180f219f-97a8-486d-99d9-ed631fe4fc57",
      "resource": {
        "resourceType": "Composition",
        "id": "180f219f-97a8-486d-99d9-ed631fe4fc57",
        "status": "final",
        "type": {
          "coding": [
            {
              "system": "http://loinc.org",
              "code": "34133-9",
              "display": "Summary of Episode Note"
            }
          ]
        },
        "subject": {
          "reference": "Patient/example"
        },
        "section": [
          {
            "title": "Allergies",
            "entry": [
              {
                "reference": "AllergyIntolerance/123"
              }
            ]
          }
        ]
      }
    }
  ]
}
```

```

    },
    {
      "fullUrl": "http://example.org/fhir/Patient/example",
      "resource": {
        "resourceType": "Patient",
        "id": "example",
        "name": [
          {
            "family": "Smith",
            "given": ["John"]
          }
        ]
      }
    },
    {
      "fullUrl": "http://example.org/fhir/AllergyIntolerance/123",
      "resource": {
        "resourceType": "AllergyIntolerance",
        "id": "123",
        "patient": {
          "reference": "Patient/example"
        },
        "code": {
          "coding": [
            {
              "system": "http://snomed.info/sct",
              "code": "418689008",
              "display": "Allergy to penicillin"
            }
          ]
        }
      }
    }
  ]
}

```

Behavior

The \$document operation:

1. Takes the specified Composition resource as the foundation for the document
2. Identifies and retrieves all resources directly referenced by the Composition
3. Packages the Composition and all referenced resources into a Bundle of type "document"

4. Stores the generated document bundle in the datastore when the `persist` parameter is set to `true`
5. Identifies and retrieves resources indirectly referenced by the Composition for comprehensive document generation

The `$document` operation currently supports retrieving resource references in the following format:

1. `GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Resource/id`
2. `Resource/id`

Unsupported resource references within the Composition resource will be filtered out from the generated document.

Error Handling

The operation handles the following error conditions:

- 400 Bad Request: Invalid `$document` operation (non-conformant request) or if the resulting document fails FHIR validation due to filtered out references when `persist` is set to `true`
- 404 Not Found: Composition resource not found

For more information about the `$document` operation specification, see the [FHIR R4 Composition \\$document](#) documentation.

Permanently Removing Resources with `$erase`

AWS HealthLake supports the `$erase` operation, enabling permanent deletion of a specific resource and its historical versions. This operation is particularly useful when you need to:

- Permanently remove individual resources
- Delete specific version histories
- Manage individual resource lifecycles
- Comply with specific data removal requirements

Usage

The \$erase operation can be invoked at two levels:

Resource Instance Level

```
POST [base]/[ResourceType]/[ID]/$erase?deleteAuditEvent=true
```

Version-Specific Level

```
POST [base]/[ResourceType]/[ID]/_history/[VersionID]/$erase
```

Parameters

Parameter	Type	Required	Default	Description
deleteAuditEvent	boolean	No	false	When true, deletes associated audit events

Examples

Example Request

```
POST [base]/Patient/example-patient/$erase?deleteAuditEvent=true
```

Example Response

```
{
  "resourceType": "OperationOutcome",
  "id": "erase-job",
  "issue": [
    {
      "severity": "information",
      "code": "informational",
      "diagnostics": "Erase job started successfully. Job ID: 12345678-1234-1234-1234-123456789012"
    }
  ]
}
```

```
]
}
```

Job Status

To check the status of an erase job:

```
GET [base]/$erase/[jobId]
```

The operation returns job status information:

```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "jobId",
      "valueString": "12345678-1234-1234-1234-123456789012"
    },
    {
      "name": "jobStatus",
      "valueString": "COMPLETED"
    },
    {
      "name": "totalResourcesDeleted",
      "valueInteger": 1
    },
    {
      "name": "startTime",
      "valueInstant": "2023-06-15T10:00:00Z"
    },
    {
      "name": "endTime",
      "valueInstant": "2023-06-15T10:05:23Z"
    }
  ]
}
```

Behavior

The \$erase operation:

1. Processes asynchronously to ensure data integrity

2. Maintains ACID transactions
3. Provides job status tracking
4. Permanently removes the specified resource and its versions
5. Includes comprehensive audit logging of deletion activities
6. Supports selective deletion of audit events

Audit Logging

The `$erase` operation logs as `DeleteResource` with user ID, timestamp, and resource details.

Limitations

- Erased resources will not appear in search responses
- Resources being erased may be temporarily inaccessible during processing
- Storage metering is adjusted immediately as resources are permanently removed

Getting patient data with `Patient/$everything`

The `Patient/$everything` operation is used to query a FHIR `Patient` resource, along with any other resources related to that `Patient`. The operation can be used to provide a patient with access to their entire record or for a provider to perform a bulk data download related to a patient. HealthLake supports `Patient/$everything` for a specific patient `id`.

`Patient/$everything` is a FHIR REST API operation that can be invoked as shown in the examples below.

GET request

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id/  
$everything
```

Note

Resources in response are sorted by resource type and resource `id`.
Response is always populated with `Bundle.total`.

Patient/\$everything parameters

HealthLake supports the following query parameters

Parameter	Details
start	Get all Patient data after a specified start date.
end	Get all Patient data before a specified end date.
since	Get all Patient data updated after a specified date.
_type	Get Patient data for specific resource types.
_count	Get Patient data and specify page size.

Example - Get all patient data after a specified start date

Patient/\$everything can use the start filter to query only data after a specific date.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id/
$everything?start=2024-03-15T00:00:00.000Z
```

Example - Get all Patient data before a specified end date

Patient \$everything can use the end filter to only query data before a specific date.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id/
$everything?end=2024-03-15T00:00:00.000Z
```

Example - Get all Patient data updated after a specified date

Patient/\$everything can use the since filter to query only data updated after a specific date.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id/
$everything?since=2024-03-15T00:00:00.000Z
```

Example - Get Patient data for specific resource types

Patient \$everything can use the _type filter to specify specific resource types to be included in the response. Multiple resource types can be specified in a comma separated list.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id/
$everything?_type=Observation,Condition
```

Example - Get Patient data and specify page size

Patient \$everything can use the `_count` to set the page size.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/id/
$everything?_count=15
```

Patient/\$everything start and end attributes

HealthLake supports the following resource attributes for the Patient/ \$everything start and end query parameters.

Resource	Resource Element
Account	Account.servicePeriod.start
AdverseEvent	AdverseEvent.date
AllergyIntolerance	AllergyIntolerance.recordedDate
Appointment	Appointment.start
AppointmentResponse	AppointmentResponse.start
AuditEvent	AuditEvent.period.start
Basic	Basic.created
BodyStructure	NO_DATE
CarePlan	CarePlan.period.start

Resource	Resource Element
CareTeam	CareTeam.period.start
ChargeItem	ChargeItem.occurrenceDateTime, ChargeItem.occurrencePeriod.start, ChargeItem.occurrenceTiming.event
Claim	Claim.billablePeriod.start
ClaimResponse	ClaimResponse.created
ClinicalImpression	ClinicalImpression.date
Communication	Communication.sent
CommunicationRequest	CommunicationRequest.occurrenceDateTime, CommunicationRequest.occurrencePeriod.start
Composition	Composition.date
Condition	Condition.recordedDate
Consent	Consent.dateTime
Coverage	Coverage.period.start
CoverageEligibilityRequest	CoverageEligibilityRequest.created
CoverageEligibilityResponse	CoverageEligibilityResponse.created

Resource	Resource Element
DetectedIssue	DetectedIssue.identified
DeviceRequest	DeviceRequest.authoredOn
DeviceUseStatement	DeviceUseStatement.recordedOn
DiagnosticReport	DiagnosticReport.effective
DocumentManifest	DocumentManifest.created
DocumentReference	DocumentReference.context.period.start
Encounter	Encounter.period.start
EnrollmentRequest	EnrollmentRequest.created
EpisodeOfCare	EpisodeOfCare.period.start
ExplanationOfBenefit	ExplanationOfBenefit.billablePeriod.start
FamilyMemberHistory	NO_DATE
Flag	Flag.period.start
Goal	Goal.statusDate
Group	NO_DATE

Resource	Resource Element
ImagingStudy	ImagingStudy.started
Immunization	Immunization.recorded
ImmunizationEvaluation	ImmunizationEvaluation.date
ImmunizationRecommendation	ImmunizationRecommendation.date
Invoice	Invoice.date
List	List.date
MeasureReport	MeasureReport.period.start
Media	Media.issued
MedicationAdministration	MedicationAdministration.effective
MedicationDispense	MedicationDispense.whenPrepared
MedicationRequest	MedicationRequest.authoredOn
MedicationStatement	MedicationStatement.dateAsserted

Resource	Resource Element
Molecular Sequence	NO_DATE
Nutrition Order	NutritionOrder.dateTime
Observation	Observation.effective
Patient	NO_DATE
Person	NO_DATE
Procedure	Procedure.performed
Provenance	Provenance.occurredPeriod.start, Provenance.occurredDateTime
QuestionnaireResponse	QuestionnaireResponse.authored
RelatedPerson	NO_DATE
RequestGroup	RequestGroup.authoredOn
ResearchSubject	ResearchSubject.period
RiskAssessment	RiskAssessment.occurrenceDateTime, RiskAssessment.occurrencePeriod.start
Schedule	Schedule.planningHorizon
ServiceRequest	ServiceRequest.authoredOn

Resource	Resource Element
Specimen	Specimen.receivedTime
SupplyDelivery	SupplyDelivery.occurrenceDateTime, SupplyDelivery.occurrencePeriod.start, SupplyDelivery.occurrenceTiming.event
SupplyRequest	SupplyRequest.authoredOn
VisionPrescription	VisionPrescription.dateWritten

Retrieving ValueSet Codes with \$expand

AWS HealthLake now supports the \$expand operation for ValueSets that were ingested by you as a customer, enabling you to retrieve the complete list of codes contained within those ValueSet resource(s). This operation is particularly useful when you need to:

- Retrieve all possible codes for validation purposes
- Display available options in user interfaces
- Perform comprehensive code lookups within a specific terminology context

Usage

The \$expand operation can be invoked on ValueSet resources using both GET and POST methods:

Supported Operations

```
GET/POST [base]/ValueSet/[id]/$expand
GET [base]/ValueSet/$expand?url=http://example.com
POST [base]/ValueSet/$expand
```

Supported Parameters

HealthLake supports a subset of FHIR R4 \$expand parameters:

Parameter	Type	Required	Description
url	uri	No	Canonical URL of the ValueSet to expand
id	id	No	ValueSet resource id to expand (for GET or POST operations)
filter	string	No	Filter the code expansion result
count	integer	No	Number of codes to return
offset	integer	No	Number of matching codes to skip before returning. Applies after filtering and only to the matching codes, not to the full, unfiltered contents of the original ValueSet

Examples

GET Request by ID

```
GET [base]/ValueSet/example-valueset/$expand
```

GET Request by URL with Filter

```
GET [base]/ValueSet/$expand?url=http://example.com/ValueSet/my-valueset&filter=male&count=5
```

POST Request with Parameters (by ID)

```
POST [base]/ValueSet/example-valueset/$expand
Content-Type: application/fhir+json

{
  "resourceType": "Parameters",
  "parameter": [
    {
```

```
    "name": "count",
    "valueInteger": 10
  },
  {
    "name": "filter",
    "valueString": "admin"
  }
]
```

POST Request with Parameters (by URL)

```
POST [base]/ValueSet/$expand
Content-Type: application/fhir+json

{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "url",
      "valueUri": "http://hl7.org/fhir/ValueSet/administrative-gender"
    },
    {
      "name": "count",
      "valueInteger": 10
    }
  ]
}
```

Sample Response

The operation returns a ValueSet resource with an expansion element containing the expanded codes:

```
{
  "resourceType": "ValueSet",
  "id": "administrative-gender",
  "status": "active",
  "expansion": {
    "identifier": "urn:uuid:12345678-1234-1234-1234-123456789abc",
    "timestamp": "2024-01-15T10:30:00Z",
```

```
{
  "total": 4,
  "parameter": [
    {
      "name": "count",
      "valueInteger": 10
    }
  ],
  "contains": [
    {
      "system": "http://hl7.org/fhir/administrative-gender",
      "code": "male",
      "display": "Male"
    },
    {
      "system": "http://hl7.org/fhir/administrative-gender",
      "code": "female",
      "display": "Female"
    },
    {
      "system": "http://hl7.org/fhir/administrative-gender",
      "code": "other",
      "display": "Other"
    },
    {
      "system": "http://hl7.org/fhir/administrative-gender",
      "code": "unknown",
      "display": "Unknown"
    }
  ]
}
```

The response includes:

- `expansion.total`: Total number of codes in the expanded ValueSet
- `expansion.contains`: Array of expanded codes with their system, code, and display values
- `expansion.parameter`: Parameters used in the expansion request

For more information about the `$expand` operation specification, see the [FHIR R4 ValueSet \\$expand](#) documentation.

Exporting HealthLake data with FHIR \$export

You can export data in bulk from your HealthLake data store using the FHIR \$export operation. HealthLake supports FHIR \$export using POST and GET requests. To make an export request with POST, you must have a IAM user, group, or role with the required permissions, specify \$export as part of the request, and include desired parameters in the request body.

Note

All HealthLake export requests made using FHIR \$export are returned in ndjson format and exported to an Amazon S3 bucket, where each Amazon S3 object contains only a single FHIR resource type.

You can queue export requests per the AWS account service quotas. For more information, see [Service quotas](#).

HealthLake supports the following three types of bulk export endpoint requests.

HealthLake bulk \$export types

Export type	Description	Syntax
System	Export all data from the HealthLake FHIR server.	POST <code>https://healthlake. . <i>region</i>.amazonaws.com/dat astore/ <i>datastoreId</i> /r4/\$export</code>
All patients	Export all data relating to all patients including resource types associated with the Patient resource type.	POST <code>https://healthlake. . <i>region</i>.amazonaws.com/dat astore/ <i>datastoreId</i> /r4/Patient/\$export</code> GET <code>https://healthlake. <i>region</i>.amazonaw s.com/datastore/ <i>datastoreId</i> /r4/Patie nt/\$export</code>
Group of patients	Export all data relating to a group of patients specified with a Group ID.	POST <code>https://healthlake . <i>region</i>.amazonaws.com/dat</code>

Export type	Description	Syntax
		<pre>astore/ <i>datastoreId</i> /r4/Group/ <i>id</i>/\$export</pre> <pre>GET https://healthlake. <i>region</i>.amazonaws.com/datastore/ <i>datastoreId</i> /r4/Group / <i>id</i>/\$export</pre>

Before you begin

Meet the following requirements to make an export request using the FHIR REST API for HealthLake.

- You must have set up a user, group, or role that has the necessary permissions to make the export request. To learn more, see [Authorizing an \\$export request](#).
- You must have created a service role that grants HealthLake access to the Amazon S3 bucket to which you want your data to be exported. The service role must also specify HealthLake as the service principal. For more information about setting up permissions, see [Setting up permissions for export jobs](#).

Authorizing an \$export request

To make a successful export request using the FHIR REST API, authorize your user, group, or role using either IAM or OAuth2.0. You must also have a service role.

Authorizing a request using IAM

When you make an \$export request, the user, group, or role must have IAM actions included in the policy. For more information, see [Setting up permissions for export jobs](#).

Authorizing a request using SMART on FHIR (OAuth 2.0)

When you make an \$export request on a SMART on FHIR-enabled HealthLake data store, you must have the appropriate scopes assigned. For more information, see [SMART on FHIR resource scopes for HealthLake](#).

Note

FHIR `$export` with GET requests require the same authentication method or bearer token (in the case of SMART on FHIR) to request the export and retrieve files. Files exported using FHIR `$export` with GET are available for download for 48 hours.

Making an `$export` request

This section describes the required steps you must take when making an export request using the FHIR REST API.

To avoid accidental charges on your AWS account, we recommend testing your requests by making a POST request without supplying the `$export` syntax.

To make the request, you must do the following:

1. Specify `$export` in the POST request URL for a supported endpoint.
2. Specify the required header parameters.
3. Specify a request body that defines the required parameters.

Step 1: Specify `$export` in the POST request URL for a supported [endpoint](#).

HealthLake supports three types of bulk export endpoint requests. To make a bulk export request, you must make a POST-based request on one of the three supported endpoints. The following examples demonstrate where to specify `$export` in the request URL.

- POST `https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/$export`
- POST `https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Patient/$export`
- POST `https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/Group/id/$export`

You can use the following supported search parameters in the POST request string.

Supported search parameters

HealthLake supports the following search modifiers in bulk export requests.

The following examples include special characters which must be encoded prior to submitting your request.

Name	Required?	Description	Example
<code>_outputFormat</code>	No	The format for the requested Bulk Data files to be generated . Accepted values are <code>application/fhir+ndjson</code> , <code>application/ndjson</code> , <code>ndjson</code> .	
<code>_type</code>	No	A string of comma delimited FHIR resource types that you want included in your export job. We recommend including <code>_type</code> because this can have a cost implication when all resources are exported.	<code>&_type=M edicationS tatement, Observation</code>
<code>_since</code>	No	Resource types modified on or after the date time stamp. If a resource type does <i>not</i> have a last updated time they will be included in your response.	<code>&_since=2 024-05-09 T00%3A00%3A00Z</code>

Step 2: Specify the required header parameters

To make an export request using the FHIR REST API, you must specify the following header parameters.

- Content-Type: `application/fhir+json`
- Prefer: `respond-async`

Next, you must specify the required elements in the request body.

Step 3: Specify a request body that defines the required parameters.

The export request also requires a body in JSON format. The body can include the following parameters.

Key	Required?	Description	Value
DataAccessRoleArn	Yes	An ARN of a HealthLake service role. The service role used must specify HealthLake as the service principal.	<code>arn:aws:iam:: 444455556666 :role/your-healthlake-service-role</code>
JobName	No	The name of the export request.	<code>your-export-job-name</code>
S3Uri	Yes	Part of an <code>OutputDataConfig</code> key. The S3 URI of the destination bucket where your exported data will be downloaded.	<code>s3://amzn-s3-demo-bucket/ EXPORT-JOB /</code>
KmsKeyId	Yes	Part of an <code>OutputDataConfig</code> key. The ARN of the AWS KMS key used to secure the Amazon S3 bucket.	<code>arn:aws:kms: region-of-bucket :123456789012 :key/1234abcd-12ab-34cd</code>

Key	Required?	Description	Value
			-56ef-1234567890ab

Example Body of an export request made using the FHIR REST API

To make an export request by using the FHIR REST API, you must specify a body, as shown in the following.

```
{
  "DataAccessRoleArn": "arn:aws:iam::444455556666:role/your-healthlake-service-role",
  "JobName": "your-export-job",
  "OutputDataConfig": {
    "S3Configuration": {
      "S3Uri": "s3://amzn-s3-demo-bucket/EXPORT-JOB",
      "KmsKeyId": "arn:aws:kms:region-of-
bucket:444455556666:key/1234abcd-12ab-34cd-56ef-1234567890ab"
    }
  }
}
```

When your request is successful, you will receive the following response.

Response Header

```
content-location: https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/
export/your-export-request-job-id
```

Response Body

```
{
  "datastoreId": "your-data-store-id",
  "jobStatus": "SUBMITTED",
  "jobId": "your-export-request-job-id"
}
```

Managing your export request

After making a successful export request, you can manage the request using `$export` to describe the status of a current export request, and `$export` to cancel a current export request.

When you cancel an export request using the REST API, you are only billed for the portion of the data that was exported up to the time you submitted the cancel request.

The following topics describe how you can get the status on or cancel a current export request.

Canceling an export request

To cancel an export request, make a DELETE request and supply the job ID in the request URL.

```
DELETE https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/export/your-export-request-job-id
```

When your request is successful, you receive the following.

```
{
  "exportJobProperties": {
    "jobId": "your-original-export-request-job-id",
    "jobStatus": "CANCEL_SUBMITTED",
    "datastoreId": "your-data-store-id"
  }
}
```

When your request is not successful, you receive the following.

```
{
  "resourceType": "OperationOutcome",
  "issue": [
    {
      "severity": "error",
      "code": "not-supported",
      "diagnostics": "Interaction not supported."
    }
  ]
}
```

Describing an export request

To get the status of an export request, make a GET request by using export and your **export-request-job-id**.

```
GET https://healthlake.region.amazonaws.com/datastore/datastoreId/r4/export/your-export-request-id
```

The JSON response will contain an `ExportJobProperties` object. It may contain the following key:value pairs.

Name	Required?	Description	Value
<code>DataAccessRoleArn</code>	No	An ARN of a HealthLake service role. The service role used must specify HealthLake as the service principal.	<code>arn:aws:iam:: 444455556666 :role/your-healthlake-service-role</code>
<code>SubmitTime</code>	No	The date time an export job was submitted.	<code>Apr 21, 2023 5:58:02</code>
<code>EndTime</code>	No	The time an export job was completed.	<code>Apr 21, 2023 6:00:08 PM</code>
<code>JobName</code>	No	The name of the export request.	<code>your-export-job-name</code>
<code>JobStatus</code>	No		Valid values are: SUBMITTED IN_PROGRESS COMPLETED _WITH_ERRORS COMPLETED FAILED
<code>S3Uri</code>	Yes	Part of an OutputDataConfig object. The Amazon S3 URI of the destination bucket where your exported data will be downloaded.	<code>s3://amzn-s3-demo-bucket/ EXPORT-JOB /</code>

Name	Required?	Description	Value
KmsKeyId	Yes	Part of an OutputDataConfig object. The ARN of the AWS KMS key used to secure the Amazon S3 bucket.	arn:aws:kms: region-of-bucket:123456789012 :key/ 1234abcd-12ab-34cd-56ef-1234567890ab

Example : Body of a describe export request made using the FHIR REST API

When successful, you will get the following JSON response.

```
{
  "exportJobProperties": {
    "jobId": "your-export-request-id",
    "jobName": "your-export-job",
    "jobStatus": "SUBMITTED",
    "submitTime": "Apr 21, 2023 5:58:02 PM",
    "endTime": "Apr 21, 2023 6:00:08 PM",
    "datastoreId": "your-data-store-id",
    "outputDataConfig": {
      "s3Configuration": {
        "S3Uri": "s3://amzn-s3-demo-bucket/EXPORT-JOB",
        "KmsKeyId": "arn:aws:kms:region-of-bucket:444455556666:key/1234abcd-12ab-34cd-56ef-1234567890ab"
      }
    },
    "DataAccessRoleArn": "arn:aws:iam::444455556666:role/your-healthlake-service-role",
  }
}
```

Retrieving Concept Details with \$lookup

AWS HealthLake now supports the \$lookup operation for CodeSystem resources, enabling you to retrieve details about a specific concept in a code system by providing identifying information such as its code. This operation is particularly useful when you need to:

- Retrieve detailed information about specific medical codes

- Validate code meanings and properties
- Access concept definitions and relationships
- Support clinical decision-making with accurate terminology data

Usage

The `$lookup` operation can be invoked on CodeSystem resources using both GET and POST methods:

Supported Operations

```
GET [base]/CodeSystem/$lookup?system=http://snomed.info/sct&code=73211009&version=20230901
POST [base]/CodeSystem/$lookup
```

Supported Parameters

HealthLake supports a subset of FHIR R4 `$lookup` parameters:

Parameter	Type	Required	Description
code	code	Yes	The concept code you're looking up (e.g., "71620000" in SNOMED CT)
system	uri	Yes	The canonical URL of the code system (e.g., " http://snomed.info/sct ")
version	string	No	Specific version of the code system

Examples

GET Request

```
GET [base]/CodeSystem/$lookup?system=http://snomed.info/sct&code=71620000&version=2023-09
```

POST Request

```
POST [base]/CodeSystem/$lookup
Content-Type: application/fhir+json

{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "system",
      "valueUri": "http://snomed.info/sct"
    },
    {
      "name": "code",
      "valueCode": "71620000"
    },
    {
      "name": "version",
      "valueString": "2023-09"
    }
  ]
}
```

Sample Response

The operation returns a Parameters resource containing the concept details:

```
{
  "resourceType": "Parameters",
  "parameter": [{
    "name": "name",
    "valueString": "SNOMED CT Fractures"
  },
  {
    "name": "version",
    "valueString": "2023-09"
  },
  {
    "name": "display",
    "valueString": "Fracture of femur"
  },
  {
    "name": "property",
    "part": [{
      "name": "code",
```

```
        "valueCode": "child"
      },
      {
        "name": "value",
        "valueCode": "263225007"
      },
      {
        "name": "description",
        "valueString": "Fracture of neck of femur"
      }
    ]
  },
  {
    "name": "property",
    "part": [{
      "name": "code",
      "valueCode": "child"
    },
    {
      "name": "value",
      "valueCode": "263227004"
    },
    {
      "name": "description",
      "valueString": "Fracture of shaft of femur"
    }
  ]
}
}
```

Response Parameters

The response includes the following parameters when available:

Parameter	Type	Description
name	string	Name of the code system
version	string	Version of the code system
display	string	Display name of the concept

Parameter	Type	Description
designation	BackboneElement	Additional representations for this concept.
property	BackboneElement	Additional properties of the concept (definition, relationships, etc.)

Behavior

The \$lookup operation:

1. Validates the required parameters (code and system)
2. Searches for the concept within the specified code system stored in the datastore
3. Returns detailed concept information including display name, designations, and properties.
4. Supports version-specific lookups when the version parameter is provided
5. Operates only on code systems explicitly stored in the HealthLake datastore

Error Handling

The operation handles the following error conditions:

- 400 Bad Request: Invalid \$lookup operation (non-conformant request or missing required parameters)
- 404 Not Found: Code system not found or code not found in the specified code system

Caveats

For this release, the following are not supported:

- \$lookup operation by calling external terminology servers
- \$lookup operation on CodeSystems managed by HealthLake but not explicitly stored in the datastore

For more information about the \$lookup operation specification, see the [FHIR R4 CodeSystem \\$lookup](#) documentation.

Removing Patient Compartment Resources with \$purge

AWS HealthLake supports the \$purge operation, enabling permanent deletion of all resources within a patient's compartment. This operation is particularly useful when you need to:

- Remove all data associated with a patient
- Comply with patient data removal requests
- Manage patient data lifecycle
- Execute comprehensive patient record cleanup

Usage

The \$purge operation can be invoked on Patient resources:

```
POST [base]/Patient/[ID]/$purge?deleteAuditEvent=true
```

Parameters

Parameter	Type	Required	Default	Description
deleteAuditEvent	boolean	No	false	When true, deletes associated audit events
_since	string	No	Datastore creation time	When entered, selects the starting cutoff time to find resources based on their lastModified time. Cannot be used with start or end
start	string	No	Datastore creation time	When entered, selects the cutoff time to find resources based on their lastModified time. Can be used with end
end	string	No	Job submission time	When entered, selects the ending cutoff time to find resources based on their lastModified time

Examples

Example Request

```
POST [base]/Patient/example-patient/$purge?deleteAuditEvent=true
```

Example Response

```
{
  "resourceType": "OperationOutcome",
  "id": "purge-job",
  "issue": [
    {
      "severity": "information",
      "code": "informational",
      "diagnostics": "Purge job started successfully. Job ID:
12345678-1234-1234-1234-123456789012"
    }
  ]
}
```

Job Status

To check the status of a purge job:

```
GET [base]/$purge/[jobId]
```

The operation returns job status information:

```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "jobId",
      "valueString": "12345678-1234-1234-1234-123456789012"
    },
    {
      "name": "jobStatus",
      "valueString": "COMPLETED"
    }
  ]
}
```

```
    },  
    {  
      "name": "totalResourcesDeleted",  
      "valueInteger": 1256  
    },  
    {  
      "name": "startTime",  
      "valueInstant": "2023-06-15T10:00:00Z"  
    },  
    {  
      "name": "endTime",  
      "valueInstant": "2023-06-15T10:05:23Z"  
    }  
  ]  
}
```

Behavior

The \$purge operation:

1. Processes asynchronously to handle multiple resources
2. Maintains ACID transactions for data integrity
3. Provides job status tracking with resource deletion counts
4. Permanently removes all resources in the patient compartment
5. Includes comprehensive audit logging of deletion activities
6. Supports selective deletion of audit events

Audit Logging

The \$purge operation logs as StartFHIRBulkDeleteJob and DescribeFHIRBulkDeleteJob with detailed operation information.

Limitations

- Purged resources will not appear in search responses
- Resources being purged may be temporarily inaccessible during processing
- All resources in the patient compartment are permanently removed

Validating FHIR Resources with \$validate

AWS HealthLake now supports the \$validate operation for FHIR resources, enabling you to validate a resource against the FHIR specification and check its conformance to a specified profile or base resource definition without performing any storage operations. This operation is particularly useful when you need to:

- Validate FHIR CMS compliance requirements
- Test resources prior to using them in production
- Provide real-time validation feedback as users edit clinical data
- Reduce invalid data submissions to create and update APIs

Usage

The \$validate operation can be invoked on FHIR resources using POST methods:

Supported Operations

```
POST [base]/[type]/[id]/$validate
POST [base]/[type]/$validate
```

Supported Payloads

Parameters resource

HealthLake supports the following FHIR \$validate parameters:

Parameter	Type	Required	Description
resource	Resource	Yes	The resource to be validated
profile	canonical	No	Canonical URL of the profile to validate against
mode	code	No	Validation mode: create, or update

Direct resource with Query Parameters

Parameter	Type	Required	Description
profile	canonical	No	Canonical URL of the profile to validate against
mode	code	No	Validation mode: create, or update

Examples

POST Request for Resource with ID and Parameters payload

```
POST [base]/Patient/example-patient/$validate
```

```
Content-Type: application/fhir+json
```

```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "resource",
      "resource": {
        "resourceType": "Patient",
        "id": "example-patient",
        "name": [
          {
            "family": "Smith",
            "given": ["John"]
          }
        ],
        "gender": "male",
        "birthDate": "1990-01-01"
      }
    },
    {
      "name": "profile",
      "valueCanonical": "http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient"
    },
    {
      "name": "mode",
      "valueString": "create"
    }
  ]
}
```

```
]
}
```

POST Request for Resource Type and Parameters payload

POST [base]/Patient/\$validate

Content-Type: application/fhir+json

```
{
  "resourceType": "Parameters",
  "parameter": [
    {
      "name": "resource",
      "resource": {
        "resourceType": "Patient",
        "name": [
          {
            "family": "Doe",
            "given": ["Jane"]
          }
        ],
        "gender": "female",
        "birthDate": "1985-05-15"
      }
    },
    {
      "name": "profile",
      "valueCanonical": "http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient"
    },
    {
      "name": "mode",
      "valueString": "update"
    }
  ]
}
```

POST Request for Resource with ID and direct resource payload

POST [base]/Patient/example-patient/\$validate?profile=<http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient>&mode=create

Content-Type: application/fhir+json

```
{
  "resourceType": "Patient",
  "id": "example-patient",
  "name": [
    {
      "family": "Smith",
      "given": ["John"]
    }
  ],
  "gender": "male",
  "birthDate": "1990-01-01"
}
```

POST Request for Resource Type and direct resource payload

```
POST [base]/Patient/$validate?profile=http://hl7.org/fhir/us/core/StructureDefinition/us-core-patient&mode=create
Content-Type: application/fhir+json
```

```
{
  "resourceType": "Patient",
  "id": "example-patient",
  "name": [
    {
      "family": "Smith",
      "given": ["John"]
    }
  ],
  "gender": "male",
  "birthDate": "1990-01-01"
}
```

Sample Response

The operation returns an `OperationOutcome` resource with validation results:

```
{
  "resourceType": "OperationOutcome",
  "issue": [
    {
      "severity": "information",
```

```
    "code": "informational",
    "diagnostics": "Validation successful"
  }
]
```

Sample Response with Validation Errors

```
{
  "resourceType": "OperationOutcome",
  "issue": [
    {
      "severity": "error",
      "code": "required",
      "details": {
        "text": "Missing required element"
      },
      "diagnostics": "Patient.identifier is required by the US Core Patient profile",
      "location": [
        "Patient.identifier"
      ]
    },
    {
      "severity": "warning",
      "code": "code-invalid",
      "details": {
        "text": "Invalid code value"
      },
      "diagnostics": "The provided gender code is not from the required value set",
      "location": [
        "Patient.gender"
      ]
    }
  ]
}
```

Behavior

The `$validate` operation:

1. Validates the resource against the FHIR specification and base resource definition
2. Checks conformance to specified profiles when the `profile` parameter is provided

3. Validates based on the specified mode (create or update)
4. Returns detailed validation results including errors, warnings, and informational messages
5. Does not perform any storage operations - validation only
6. Returns HTTP 200 OK when validation can be performed, regardless of whether validation issues are found

Validation Modes

- **create:** Validates the resource as if it were being created (new resource)
- **update:** Validates the resource as if it were being updated (existing resource)

Error Handling

The operation returns:

- 200 OK: Validation was performed successfully (regardless of validation outcome)
- 400 Bad Request: Invalid request format or parameters
- 404 Not Found: Resource type or profile not found

For more information about the `$validate` operation specification, see the [FHIR R4 Resource `\$validate`](#) documentation.

Support reference for AWS HealthLake

The following supporting reference material is available for AWS HealthLake.

Note

All native HealthLake actions and data types are described in a separate reference. For more information, see the [AWS HealthLake API Reference](#).

Topics

- [AWS HealthLake endpoints and quotas](#)
- [Synthea preloaded data types for HealthLake](#)

- [AWS HealthLake sample projects](#)
- [Troubleshooting AWS HealthLake](#)
- [Using HealthLake with an AWS SDK](#)

AWS HealthLake endpoints and quotas

The following topics contain information about AWS HealthLake service endpoints and quotas.

Topics

- [Service endpoints](#)
- [Service quotas](#)

Service endpoints

A service endpoint is a URL that identifies a host and port as the entry point for a web service. Every web service request contains an endpoint. Most AWS services provide endpoints for specific Regions to enable faster connectivity. The following table lists the service endpoints for AWS HealthLake.

Region Name	Region	Endpoint	Protocol	
US East (Ohio)	us-east-2	healthlake.us-east-2.amazonaws.com	HTTPS	
		healthlake-fips.us-east-2.amazonaws.com	HTTPS	
US East (N. Virginia)	us-east-1	healthlake.us-east-1.amazonaws.com	HTTPS	
		healthlake-fips.us-east-1.amazonaws.com	HTTPS	
US West (Oregon)	us-west-2	healthlake.us-west-2.amazonaws.com	HTTPS	
		healthlake-fips.us-west-2.amazonaws.com	HTTPS	
Asia Pacific (Mumbai)	ap-south-1	healthlake.ap-south-1.amazonaws.com	HTTPS	

Region Name	Region	Endpoint	Protocol	
Asia Pacific (Sydney)	ap-southeast-2	healthlake.ap-southeast-2.amazonaws.com	HTTPS	
Europe (Ireland)	eu-west-1	healthlake.eu-west-1.amazonaws.com	HTTPS	
Europe (London)	eu-west-2	healthlake.eu-west-2.amazonaws.com	HTTPS	

Service quotas

Service quotas are defined as the maximum value for resources, actions, and items in your AWS account.

Note

For adjustable quotas, you can request a quota increase using the [Service Quotas console](#). For more information, see [Requesting a quota increase](#) in the *Service Quotas User Guide*. Sync Write API rate proportionally increases with payload size, with each 1KB increment consuming additional capacity (e.g., a 4KB payload uses 4x write capacity). Setting the optional `x-amz-fhir-history-consistency-level` header to `strong` doubles the write capacity consumption per resource.

Resources within Bundles follow standard read/write limits based on 1 KB payload size. Bundle type *transaction* consumes twice the write capacity compared to type *batch*, meaning *batch* Bundles can process twice as many resources per second as transaction Bundles.

The following table lists the default quotas for AWS HealthLake.

Name	Default	Adjustable	Description
Number of characters in a medical note	Each supported Region: 10,000	No	The maximum number of characters in an individual medical note within the DocumentReference Resource type (POST/PUT requests).
Number of concurrent StartFHIR ExportJob jobs	Each supported Region: 1	No	The maximum concurrent StartFHIRExportJob jobs.
Number of concurrent StartFHIR ImportJob jobs	Each supported Region: 1	No	The maximum concurrent StartFHIRImportJob jobs.
Number of data stores per account	Each supported Region: 10	Yes	The default maximum number of active data stores per account.
Number of files in a StartFHIR ImportJob	Each supported Region: 10,000	Yes	The maximum number of files in a StartFHIR ImportJob.
Number of resources per Bundle	Each supported Region: 500	No	The maximum number of resources allowed in a Bundle request.
Rate of CancelFHIRExportJob requests using DELETE per account	Each supported Region: 1	No	The maximum number of CancelFHIRExportJob requests using DELETE that you can make per second per account.
Rate of CreateFHIRDatastore requests per account	Each supported Region: 1	No	The maximum number of CreateFHIRDatastor

Name	Default	Adjust	Description
			e requests that you can make per minute per account.
Rate of DeleteFHIRDatastore requests per account	Each supported Region: 1	No	The maximum number of DeleteFHIRDatastore requests that you can make per minute per account.
Rate of DescribeFHIRDatastore requests per account	Each supported Region: 10	No	The maximum number of DescribeFHIRDatastore requests that you can make per second per account.
Rate of DescribeFHIRExportJob requests per account	Each supported Region: 10	No	The maximum number of DescribeFHIRExportJob requests that you can make per second per account.
Rate of DescribeFHIRExportJob requests using GET per account	Each supported Region: 10	No	The maximum number of DescribeFHIRExportJob requests using GET that you can make per second per account.
Rate of DescribeFHIRImportJob requests per account	Each supported Region: 10	No	The maximum number of DescribeFHIRImportJob requests that you can make per second per account.

Name	Default	Adjustable	Description
Rate of Discovery requests per account	Each supported Region: 10	No	The maximum number of Discovery requests that you can make per minute per account.
Rate of GET requests per account	Each supported Region: 6,000	Yes	The maximum number of GET requests that you can make per second per account.
Rate of GET requests per data store	Each supported Region: 3,000	Yes	The maximum number of GET requests that you can make per second per data store. Data stores created prior to 8/21/2023 will be limited to 100 requests per second.
Rate of GetCapabilities requests per account	Each supported Region: 10	No	The maximum number of GetCapabilities requests that you can make per second per account.
Rate of GetExportedFile requests per datastore	Each supported Region: 10	No	The maximum number of GetExportedFile requests that you can make per second per datastore.
Rate of ListFHIRDatastores requests per account	Each supported Region: 10	No	The maximum number of ListFHIRDatastores requests that you can make per second per account.

Name	Default	Adjust	Description
Rate of ListFHIRExportJobs requests per account	Each supported Region: 10	No	The maximum number of ListFHIRExportJobs requests that you can make per second per account.
Rate of ListFHIRImportJobs requests per account	Each supported Region: 10	No	The maximum number of ListFHIRImportJobs requests that you can make per second per account.
Rate of ListTagsforResource requests per account	Each supported Region: 10	No	The maximum number of ListTagsforResource requests that you can make per second per account.
Rate of StartFHIRExportJob requests per account	Each supported Region: 1	No	The maximum number of StartFHIRExportJob requests that you can make per second per account.
Rate of StartFHIRExportJob requests using GET per account	Each supported Region: 1	No	The maximum number of StartFHIRExportJob requests using GET that you can make per second per account.
Rate of StartFHIRExportJob requests using POST per account	Each supported Region: 1	No	The maximum number of StartFHIRExportJob requests using POST that you can make per second per account.

Name	Default	Adjustable	Description
Rate of StartFHIRImportJob requests per account	Each supported Region: 25	No	The maximum number of StartFHIRImportJob requests that you can make per second per account.
Rate of TagResource requests per account	Each supported Region: 10	No	The maximum number of TagResource requests that you can make per second per account.
Rate of UntagResource requests per account	Each supported Region: 10	No	The maximum number of UntagResource requests that you can make per second per account.
Rate of WRITE requests per account	Each supported Region: 6,000	Yes	The maximum number of CREATE UPDATE DELETE requests that you can make per second per account.
Rate of WRITE requests per data store	Each supported Region: 3,000	Yes	The maximum number of CREATE UPDATE DELETE requests that you can make per second per data store. Data stores created prior to 8/21/2023 will be limited to 300 requests per second.

Name	Default	Adjustable	Description
Rate of search requests using GET per account	Each supported Region: 200	Yes	The maximum number of search requests using GET that you can make per second per account.
Rate of search requests using GET per data store	Each supported Region: 100	Yes	The maximum number of search requests using GET that you can make per second per data store.
Rate of search requests using POST per account	Each supported Region: 200	Yes	The maximum number of search requests using POST that you can make per second per account.
Rate of search requests using POST per data store	Each supported Region: 100	Yes	The maximum number of search requests using POST that you can make per second per data store.
Size of individual imported file	Each supported Region: 5 Gigabytes	No	The maximum size (in GB) of an individual file included in a StartFHIR ImportJob.
Total Queued Bulk Export jobs per datastore	Each supported Region: 25	Yes	The maximum number of queued bulk export jobs per datastore at any given time.

Name	Default	Adjustable	Description
Total Queued Bulk Import jobs per datastore	Each supported Region: 25	Yes	The maximum number of queued bulk import jobs per datastore at any given time.
Total import job size	Each supported Region: 500 Gigabytes	Yes	The maximum size (in GB) of all files included in the import job.

Synthea preloaded data types for HealthLake

HealthLake supports only SYNTHEA as a preloaded data type. [Synthea](#) is a synthetic patient generator that models Patient medical history. It's hosted in an open-source Git repository that allows HealthLake to generate a FHIR R4-compliant resource Bundle so that users can test models without using actual patient data.

The following resource types are available in preloaded HealthLake data stores. For more information about preloading HealthLake data stores with Synthea data, see [Creating a HealthLake data store](#).

Note

To view a full list of HealthLake-supported FHIR R4 resources, see [FHIR R4 supported resource types for HealthLake](#).

Synthea FHIR resource types supported by HealthLake

AllergyIntolerance	Location
CarePlan	MedicationAdministration
CareTeam	MedicationRequest
Claim	Observation

Condition	Organization
Device	Patient
DiagnosticReport	Practitioner
Encounter	PractitionerRole
ExplanationofBenefit	Procedure
ImagingStudy	Provenance
Immunization	

AWS HealthLake sample projects

To further your analysis, you can use HealthLake with other AWS services as demonstrated in the following blog post examples.

HealthLake integrated analytics

- [Population health applications with AWS HealthLake – Part 1: Analytics and monitoring using Amazon QuickSight.](#)
- [Building predictive disease models using Amazon SageMaker AI with AWS HealthLake normalized data.](#)
- [Build a cognitive search and a health knowledge graph using AWS AI services.](#)

HealthLake event monitoring

- [Amazon EventBridge integration with AWS HealthLake.](#)

Troubleshooting AWS HealthLake

The following topics provide troubleshooting advice for errors and issues that you might encounter when using the AWS CLI, AWS SDKs, or HealthLake console. If you find an issue that is not listed in this section, use the **Provide feedback** button on the right sidebar of this page to report it.

Topics

- [Data store actions](#)
- [Import actions](#)
- [FHIR APIs](#)
- [NLP integrations](#)
- [SQL integrations](#)

Data store actions

Issue: *When I try to create a HealthLake data store, I receive the following error:*

```
AccessDeniedException: Insufficient Lake Formation permission(s): Required Database on Catalog
```

On November 14, 2022, HealthLake updated the required IAM permissions to create a new data store. For more information, see [Configure an IAM user or role to use HealthLake \(IAM Administrator\)](#).

Issue: *When creating a HealthLake data store using the AWS SDKs, the data store creation status returns an exception or unknown status.*

Update your AWS SDK to the latest version if your `DescribeFHIRDatastore` or `ListFHIRDatastores` API calls return an exception or unknown data store status.

Import actions

Issue: *Can I still use HealthLake if my data isn't in FHIR R4 format?*

Only FHIR R4 formatted data can be imported into a HealthLake data store. For a list of partners that can help transform existing health data to FHIR R4 format, see [AWS HealthLake Partners](#).

Issue: *Why did my FHIR import job fail?*

A successful import job will generate a folder with results (output log) in `.ndjson` format, however, individual records can fail to import. When this happens, a second `FAILURE` folder will be generated with a manifest of records that failed to import. For more information, see [Importing FHIR data with AWS HealthLake](#).

To analyze why an import job failed use the `DescribeFHIRImportJob` API to analyze the `JobProperties`. The following is recommended:

- If the status is `FAILED` and a message is present, the failures are related to job parameters such as input data size or number of input files being beyond HealthLake quotas.
- If the import job status is `COMPLETED_WITH_ERRORS`, check the manifest file, `manifest.json`, for information on which files did not import successfully.
- If the import job status is `FAILED` and a message is not present, go to the job output location to access the manifest file, `manifest.json`.

For each input file, there is failure output file with input file name for any resource that fails to import. The responses contain line number (`lineId`) corresponding to the location of input data, FHIR response object (`UpdateResourceResponse`), and status code (`statusCode`) of the response.

A sample output file might be similar to the following:

```
{
  "lineId": 3,
  "UpdateResourceResponse": {
    "jsonBlob": {
      "resourceType": "OperationOutcome",
      "issue": [
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "1 validation error detected: Value 'Patient123' at 'resourceType' failed to satisfy constraint: Member must satisfy regular expression pattern: [A-Za-z]{1,256}"
        }
      ]
    },
    "statusCode": 400
  },
  "lineId": 5,
  "UpdateResourceResponse": {
    "jsonBlob": {
      "resourceType": "OperationOutcome",
      "issue": [
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "This property must be a simple value, not a com.google.gson.JsonArray",
          "location": ["/EffectEvidenceSynthesis/name"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Unrecognised property '@telecom'",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Unrecognised property '@gender'",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Unrecognised property '@birthDate'",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Unrecognised property '@address'",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Unrecognised property '@maritalStatus'",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Unrecognised property '@multipleBirthBoolean'",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Unrecognised property '@communication'",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "warning",
          "code": "processing",
          "diagnostics": "Name should be usable as an identifier for the module by machine processing applications such as code generation [name.matches('[A-Z]([A-Za-z0-9_]){0,254}']",
          "location": ["/EffectEvidenceSynthesis"]
        },
        {
          "severity": "error",
          "code": "processing",
          "diagnostics": "Profile http://hl7.org/fhir/StructureDefinition/EffectEvidenceSynthesis, Element 'EffectEvidenceSynthesis.status':",
          "location": ["/EffectEvidenceSynthesis/status"]
        }
      ]
    }
  }
}
```

```

    minimum required = 1, but only found 0", "location": ["EffectEvidenceSynthesis"]},
{"severity": "error", "code": "processing", "diagnostics": "Profile
http://hl7.org/fhir/StructureDefinition/EffectEvidenceSynthesis,
Element 'EffectEvidenceSynthesis.population': minimum required
= 1, but only found 0", "location": ["EffectEvidenceSynthesis"]},
{"severity": "error", "code": "processing", "diagnostics": "Profile
http://hl7.org/fhir/StructureDefinition/EffectEvidenceSynthesis,
Element 'EffectEvidenceSynthesis.exposure': minimum required =
1, but only found 0", "location": ["EffectEvidenceSynthesis"]},
{"severity": "error", "code": "processing", "diagnostics": "Profile http://
hl7.org/fhir/StructureDefinition/EffectEvidenceSynthesis, Element
'EffectEvidenceSynthesis.exposureAlternative': minimum required
= 1, but only found 0", "location": ["EffectEvidenceSynthesis"]},
{"severity": "error", "code": "processing", "diagnostics": "Profile http://hl7.org/fhir/
StructureDefinition/EffectEvidenceSynthesis, Element 'EffectEvidenceSynthesis.outcome':
minimum required = 1, but only found 0", "location": ["EffectEvidenceSynthesis"]},
{"severity": "information", "code": "processing", "diagnostics": "Unknown
extension http://synthetichealth.github.io/synthea/disability-adjusted-
life-years", "location": ["EffectEvidenceSynthesis.extension[3]"]},
{"severity": "information", "code": "processing", "diagnostics": "Unknown extension
http://synthetichealth.github.io/synthea/quality-adjusted-life-years", "location":
["EffectEvidenceSynthesis.extension[4]"]}], "statusCode": 400}
{"lineId": 7, UpdateResourceResponse: {"jsonBlob":
{"resourceType": "OperationOutcome", "issue":
[{"severity": "error", "code": "processing", "diagnostics": "2 validation errors detected:
Value at 'resourceId' failed to satisfy constraint: Member must satisfy regular
expression pattern: [A-Za-z0-9-.]{1,64}; Value at 'resourceId' failed to satisfy
constraint: Member must have length greater than or equal to 1"}]}, "statusCode": 400}
{"lineId": 9, UpdateResourceResponse: {"jsonBlob":
{"resourceType": "OperationOutcome", "issue":
[{"severity": "error", "code": "processing", "diagnostics": "Missing required id field in
resource json"]}], "statusCode": 400}
{"lineId": 15, UpdateResourceResponse: {"jsonBlob":
{"resourceType": "OperationOutcome", "issue":
[{"severity": "error", "code": "processing", "diagnostics": "Invalid JSON found in input
file"}]}, "statusCode": 400}

```

The example above shows that there were failures on lines 3, 4, 7, 9, 15 from the corresponding input lines from input file. For each of those lines, the explanations are as follows:

- On Line 3, the response explains that resourceType provided in line 3 of input file is not valid.
- On Line 5, the response explains that there is a FHIR validation error in line 5 of input file.

- On Line 7, the response explains that there is a validation issue with `resourceId` provided as input.
- On Line 9, the response explains that input file must contain a valid resource id.
- On line 15, the response of input file is that the file is not in a valid JSON format.

FHIR APIs

Issue: *How do I implement authorization for the FHIR RESTful APIs?*

Determine the [Data store authorization strategy](#) to use.

To create SigV4 authorization using the AWS SDK for Python (Boto3), create a script similar to the following example.

```
import boto3
import requests
import json
from requests_auth_aws_sigv4 import AWSSigV4

# Set the input arguments
data_store_endpoint = 'https://healthlake.us-east-1.amazonaws.com/datastore/<datastore
id>/r4/'
resource_path = "Patient"
requestBody = {"resourceType": "Patient", "active": True, "name": [{"use":
"official", "family": "Dow", "given": ["Jen"]}, {"use": "usual", "given":
["Jen"]}], "gender": "female", "birthDate": "1966-09-01"}
region = 'us-east-1'

#Frame the resource endpoint
resource_endpoint = data_store_endpoint+resource_path
session = boto3.session.Session(region_name=region)
client = session.client("healthlake")

# Frame authorization
auth = AWSSigV4("healthlake", session=session)

# Call data store FHIR endpoint using SigV4 auth

r = requests.post(resource_endpoint, json=requestBody, auth=auth, )
print(r.json())
```

Issue: *Why am I receiving `AccessDenied` errors when using the FHIR RESTful APIs for a data store encrypted with a customer managed KMS key?*

Permissions for both customer managed keys and IAM policies are required for a user or role to access a data store. A user must have the required IAM permissions for using a customer managed key. If a user revoked or retired a grant that gave HealthLake permission to use the customer managed KMS key, HealthLake will return an `AccessDenied` error.

HealthLake must have the permission in place to access customer data, to encrypt new FHIR resources imported to a data store, and to decrypt the FHIR resources when they are requested. For more information, see [Troubleshooting AWS KMS permissions](#).

Issue: *A FHIR `POST` API operation to HealthLake using a 10MB document is returning the `413 Request Entity Too Large` error.*

AWS HealthLake has a synchronous Create and Update API limit of 5MB to avoid increased latencies and timeouts. You can ingest large documents, up to 164MB, using the Binary resource type using the Bulk Import API.

NLP integrations

Issue: *How do I turn on HealthLake's integrated natural language processing feature?*

As of November 14, 2022, the default behavior of HealthLake data stores changed.

Current data stores: All current HealthLake data stores will stop using natural language processing (NLP) on base64-encoded `DocumentReference` resources. This means that new `DocumentReference` resources will not be analyzed using NLP, and no new resources will be generated based off of text in the `DocumentReference` resource type. For existing `DocumentReference` resources, the data and resources generated via NLP remain, but they will not be updated after February 20, 2023.

New data stores: HealthLake data stores created after February 20, 2023 will *not* perform natural language processing (NLP) on base64-encoded `DocumentReference` resources.

To turn on HealthLake NLP integration, create a support case using [AWS Support Center Console](#). To create your case, log in to your AWS account, and then choose **Create case**. To learn more about creating a case and case management, see [Creating support cases and case management](#) in the *Support User Guide*.

Issue: *>How do I find DocumentReference resources that could not be processed by integrated NLP?*

If a DocumentReference resource is not valid, HealthLake provides an extension indicating a validation error instead of providing it in the integrated medical NLP output. To find DocumentReference resources that led to a validation error during NLP processing, you can use HealthLake's FHIR search function with search key **cm-decoration-status** and search value **VALIDATION_ERROR**. This search will list all DocumentReference resources that led to validation errors, along with an error message describing the nature of the error. The structure of the extension field in those DocumentReference resources with validation errors will resemble the following example.

```
"extension": [
  {
    "extension": [
      {
        "url": "http://healthlake.amazonaws.com/aws-cm/status/",
        "valueString": "VALIDATION_ERROR"
      },
      {
        "url": "http://healthlake.amazonaws.com/aws-cm/message/",
        "valueString": "Resource led to too many nested objects after NLP
operation processed the document. 10937 nested objects exceeds the limit of 10000."
      }
    ],
    "url": "http://healthlake.amazonaws.com/aws-cm/"
  }
]
```

Note

A VALIDATION_ERROR can also occur if NLP decoration creates more than 10,000 nested objects. When this happens, the document must be split into smaller documents before processing.

SQL integrations

Issue: *Why do I get a Lake Formation permissions error:*

lakeformation:PutDataLakeSettings when adding a new data lake administrator?

If your IAM user or role contains the `AWSLakeFormationDataAdmin` AWS managed policy you cannot add new data lake administrators. You will get an error containing the following:

```
User arn:aws:sts::111122223333:assumed-role/lakeformation-admin-user is not authorized
to perform: lakeformation:PutDataLakeSettings on resource: arn:aws:lakeformation:us-
east-2:111122223333:catalog:111122223333 with an explicit deny in an identity-based
policy
```

The AWS managed policy `AdministratorAccess` is required to add an IAM user or role as a AWS Lake Formation data lake administrator. If your IAM user or role also contains `AWSLakeFormationDataAdmin` the action will fail. The `AWSLakeFormationDataAdmin` AWS managed policy contains an explicit deny for the AWS Lake Formation API operation, `PutDataLakeSetting`. Even administrators with full access to AWS using the `AdministratorAccess` managed policy can be limited by the `AWSLakeFormationDataAdmin` policy.

Issue: *How do I migrate an existing HealthLake data store to use Amazon Athena SQL integration?*

HealthLake data stores created before November 14, 2022 are functional, but are not queryable in Athena using SQL. To query a preexisting data store with Athena, you must first migrate it to a new data store.

To migrate your HealthLake data to a new data store

1. Create a new data store.
2. Export the data from the pre-existing to an Amazon S3 bucket.
3. Import the data into the new data store from the Amazon S3 bucket.

Note

Exporting data to an Amazon S3 bucket incurs an extra charge. The extra charge depends on the size of the data that you export.

Issue: *When creating a new HealthLake data store for SQL integration, the data store status is not changing from `Creating`.*

If you try to create a new HealthLake data store, and your data store status is not changing from **Creating** you need to update Athena to use the AWS Glue Data Catalog. For more information, see [Upgrading to the AWS Glue Data Catalog step-by-step](#) in the *Amazon Athena User Guide*.

After successfully upgrading the AWS Glue Data Catalog, you can create a HealthLake data store.

To remove an old HealthLake data store, create a support case using [AWS Support Center Console](#). To create your case, log in to your AWS account, and then choose **Create case**. To learn more, see [Creating support cases and case management](#) in the *Support User Guide*.

Issue: *The Athena console is not working after importing data into a new HealthLake data store*

After you import data into a new HealthLake data store, the data may not be available for immediate use. This is to allow time for the data to be ingested into Apache Iceberg tables. Try again at a later time.

Issue: *How do I connect search results in Athena to other AWS services?*

When sharing your search results from Athena with other AWS services, issues can occur when you use `json_extract[1]` as part of a SQL search query. To fix this issue, you must update to CATVAR.

You might encounter this issue when trying to **Create** save results, a **Table** (static), or a **View** (dynamic).

Using HealthLake with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS CLI	AWS CLI code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples

SDK documentation	Code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS Tools for PowerShell	AWS Tools for PowerShell code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

AWS HealthLake releases

The following table shows when features and updates were released for AWS HealthLake. For more information about a release, see the linked topic.

Change	Description	Date
Patch Operation	HealthLake allows modifying specific elements of FHIR resources using JSON Patch operations without updating the entire resource.	August 18, 2025
\$validate Operation	HealthLake enables validating FHIR resources against the specification and profiles without performing storage operations, returning detailed validation results.	August 18, 2025
\$purge Operation	HealthLake includes functionality to permanently delete all resources within a patient's compartment from the datastore.	August 18, 2025
\$lookup Operation	HealthLake provides the ability to retrieve detailed information about a specific concept in a CodeSystem by providing the code and system identifier.	August 18, 2025
\$expand Operation	HealthLake now allows expanding ValueSet resources to retrieve the complete list	August 18, 2025

of codes contained within customer-ingested ValueSets.

[\\$erase Operation](#)

HealthLake now offers permanently deletion of a specific resource and all its historical versions from the datastore.

August 18, 2025

[\\$document Operation](#)

HealthLake supports generating complete clinical documents by bundling a Composition resource with all its referenced resources into a single document Bundle.

August 18, 2025

[:below Search Modifier](#)

HealthLake introduces searching for URI values that are hierarchically below the specified URI in a terminology system.

August 8, 2025

[Conditional Delete](#)

HealthLake now supports FHIR Conditional Delete, allowing healthcare organizations to delete an existing resource based on search criteria rather than by logical FHIR ID. See [Deleting FHIR resources based on conditions](#) for more information.

July 7, 2025

[Added support for new IGs](#)

AWS HealthLake has expanded its FHIR Implementation Guides (IGs) support for the following:

July 7, 2025

- US Core 7.0.0 which specifies how to use FHIR to implement the USCDI 4.0 standard
- UK Core 2.0.1 Implementation Guide to provide UK wide FHIR implementation guidance

[Bundle Type Transactions](#)

HealthLake now supports FHIR Bundle type 'Transaction', allowing healthcare organizations to submit multiple resources as a single atomic operation. This enables more efficient data exchange and integration workflows. For example, a healthcare provider can now update a patient record, medication list, and appointment in a single transaction, reducing complexity and potential errors. See [Bundling FHIR Resources](#) for more information.

April 28, 2025

[Added support for new IGs](#)

AWS HealthLake AWS
HealthLake has expanded its
FHIR Implementation Guides
(IGs) support for the following
:

April 28, 2025

- NCQA HEDIS® Implementation Guide (0.3.1):
Supports quality measurement and reporting for the Healthcare Effectiveness Data and Information Set (HEDIS).
- International Patient Summary (IPS) (2.0.0):
Enables the exchange of essential health information to support continuity of care for patients.
- Quality Measure (5.0.0):
Supports the representation and exchange of quality measure definitions and data.
- Genomics Reporting (3.0.0):
Facilitates the exchange of genomic data and reports.

[Idempotency Keys](#)

HealthLake now supports
idempotency keys for FHIR
POST operations, providing a
robust mechanism to ensure
data integrity during resource
creation. See [Idempotency
and Concurrency](#) for more
information.

April 18, 2025

[FHIR history consistency](#)

HealthLake now supports strong consistency for [history](#)-enabled data stores via the new `x-amz-fhir-history-consistency-level` header. When set to 'strong', FHIR search results include all indexed records regardless of update status. See [FHIR Search Consistency Levels](#) for more information.

April 18, 2025

[Etag and 'if-match'](#)

HealthLake now provides eTag support, allowing clients to use the 'If-Match' header to ensure idempotent updates. This helps maintain data integrity by preventing accidental overwrites during concurrent updates. This is particularly valuable in high-volume healthcare environments where multiple systems might attempt to update the same record simultaneously. See [ETag in AWS HealthLake](#) for more information.

April 18, 2025

[Conditional PUTs in Bundles](#)

HealthLake now supports conditional updates for FHIR Bundles, giving healthcare organizations more flexibility to manage and update their data. Clients can now specify criteria to conditionally create, update, or delete resources as part of a Bundle transaction. This simplifies data synchronization processes between systems and reduces the need for complex client-side logic. See [Conditional PUTs in Bundles](#) for more information.

April 18, 2025

[SMART on FHIR V2 scopes](#)

HealthLake supports SMART on FHIR V2 scopes for creating, reading, updating, deleting, and searching FHIR resources. For more information, see [SMART on FHIR resource scopes for HealthLake](#).

January 22, 2025

- SMART on FHIR V2 scopes are available to all HealthLake data stores created after 01/22/2025. If your data store was created before this date, you can submit a support ticket to have SMART on FHIR V2 scopes enabled. Create a case using [AWS Support Center Console](#). To create your case, log in to your AWS account and choose **Create case**.

[FHIR US Core Profile, version 6.1.0](#)

HealthLake supports version 6.1.0 of the FHIR US Core Profile. For more information, see [FHIR profile validations for HealthLake](#).

January 22, 2025

[FHIR \\$export with GET](#)

HealthLake supports FHIR \$export with GET. For more information, see [Exporting HealthLake data with FHIR \\$export](#).

January 22, 2025

[Refactored Developer Guide with tested code examples](#)

HealthLake introduces a refactored Developer Guide with tested code examples for native AWS CLI and AWS SDK actions. In addition, procedures are now available for all supported FHIR API interactions. For more information, see [Code examples](#) and [Managing FHIR resources](#).

December 18, 2024

[FHIR history and vread interactions](#)

HealthLake supports the FHIR history interaction for retrieving the history of a particular resource and the vread interaction for performing a version-specific read of a resource. For more information, see [Reading FHIR resource history](#).

October 25, 2024

- FHIR resource history is enabled by default to all HealthLake data stores created after 10/25/2024. If your data store was created before this date, you can submit a support ticket to have FHIR history interaction enabled. Create a case using [AWS Support Center Console](#). To create your case, log in to your AWS account and choose **Create case**.

[FHIR Patient/\\$everything operation](#)

HealthLake supports the FHIR Patient/\$everything operation for searching a Patient resource and all its related resources. Using this operation, you can access a patient's entire record or download Patient data in bulk. For more information, see [Getting patient data with Patient/\\$everything](#).

- FHIR Patient/\$everything is enabled by default to all HealthLake data stores created after 02/27/2024. If your data store was created before this date, you can submit a support ticket to have the Patient/\$everything operation enabled. Create a case using [AWS Support Center Console](#). To create your case, log in to your AWS account and choose **Create case**.

[FHIR VerificationResult resource](#)

HealthLake supports the FHIR VerificationResult resource type for describing validation requirements, sources, status, and dates for one or more elements. For more information, see [FHIR R4 resource types for HealthLake](#).

December 9, 2023

FHIR \$export operation

June 1, 2023

HealthLake supports the FHIR \$export operation for exporting health data in bulk from an HealthLake data store. For more information, see [Exporting HealthLake data with FHIR \\$export](#).

- FHIR \$export is enabled by default to all HealthLake data stores created after June 1, 2023. If your data store was created before this date, you can submit a support ticket to have the \$export operation enabled. Create a case using [AWS Support Center Console](#). To create your case, log in to your AWS account and choose **Create case**.
- HealthLake data stores created prior to 06/01/23 support only \$export job requests for system-wide exports.
- HealthLake data stores created prior to 06/01/23 do not support getting the status of an FHIR \$export using a GET request on the data store's endpoint.

SMART on FHIR support	HealthLake adds support for SMART on FHIR authorization. For more information, see SMART on FHIR support for AWS HealthLake .	May 31, 2023
FHIR profile validations	HealthLake supports FHIR profile validations for defining specific resource type definitions using constraints and/or extensions on base resource types. For more information, see Profile validations .	May 31, 2023
Asia Pacific (Mumbai) region	HealthLake is available in the Asia Pacific (Mumbai) region. For more information, see Service endpoints .	April 4, 2023
Natural language processing turned off by default	HealthLake turned off integrated natural language processing (NLP) on all data stores as of February 20, 2023. You can submit a support ticket to have integrated NLP functionality turned on. Create a case using AWS Support Center Console . To create your case, log in to your AWS account and choose Create case . To learn more about integrated NLP, see Integrating NLP with HealthLake .	February 20, 2023

-

[SQL index and query with Amazon Athena](#)

HealthLake supports querying FHIR data with SQL using Amazon Athena. For more information, see [Querying HealthLake data with Amazon Athena](#).

November 14, 2022

- SQL query functionality is enabled by default to all HealthLake data stores created after 11/14/2022. If your data store was created before this date, you can submit a support ticket to have SQL query functionality enabled. Create a case using [AWS Support Center Console](#). To create your case, log in to your AWS account and choose **Create case**.
- With SQL query functionality, IAM settings to access HealthLake must be updated. To both create HealthLake data stores and grant access to them in Athena, you must have the `AWSLakeFormationDataAdmin` managed policy added to your IAM user, group, or role. You can use the `AWSLakeFormationDataAdmin` policy to create data lake administrators and grant

	access to data stores in Athena. For more information, see Configure an IAM user or role .	
Total import job size increased	HealthLake updates the Total import job size for a StartFHIR ImportJob request to 500 GB. For more information, see Service quotas .	October 3, 2022
FHIR Bundle resource	HealthLake supports the FHIR Bundle resource type for processing multiple FHIR resources simultaneously. For more information, see Bundling FHIR resources .	August 5, 2022
Quota updates for FHIR interactions	HealthLake updates quotas for FHIR resource management interactions. For more information, see Service quotas .	July 16, 2022
FHIR _include search parameter	HealthLake adds support for the FHIR _include search parameter to return additional resources in a search request. For more information, see Advanced search parameters .	July 16, 2022
AWS HealthLake is generally available	HealthLake is generally available in all supported regions. For more information, see Service endpoints .	July 15, 2021