# AWS X-Ray

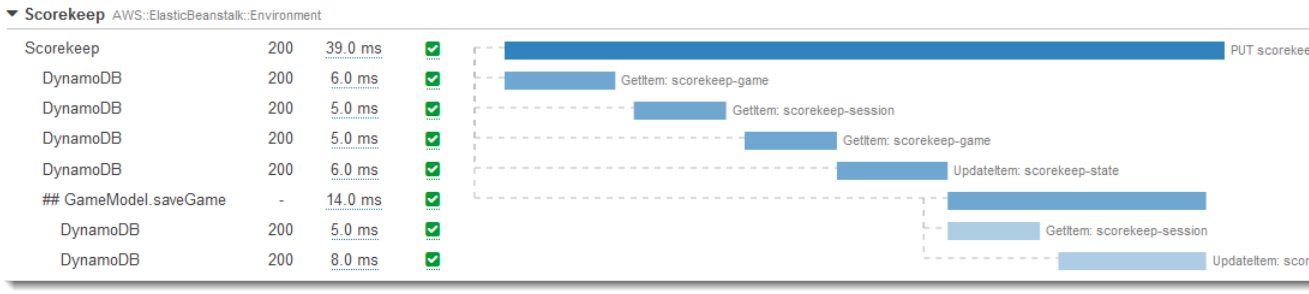
## Developer Guide

# AWS X-Ray: Developer Guide

# Table of Contents

# What is AWS X-Ray?

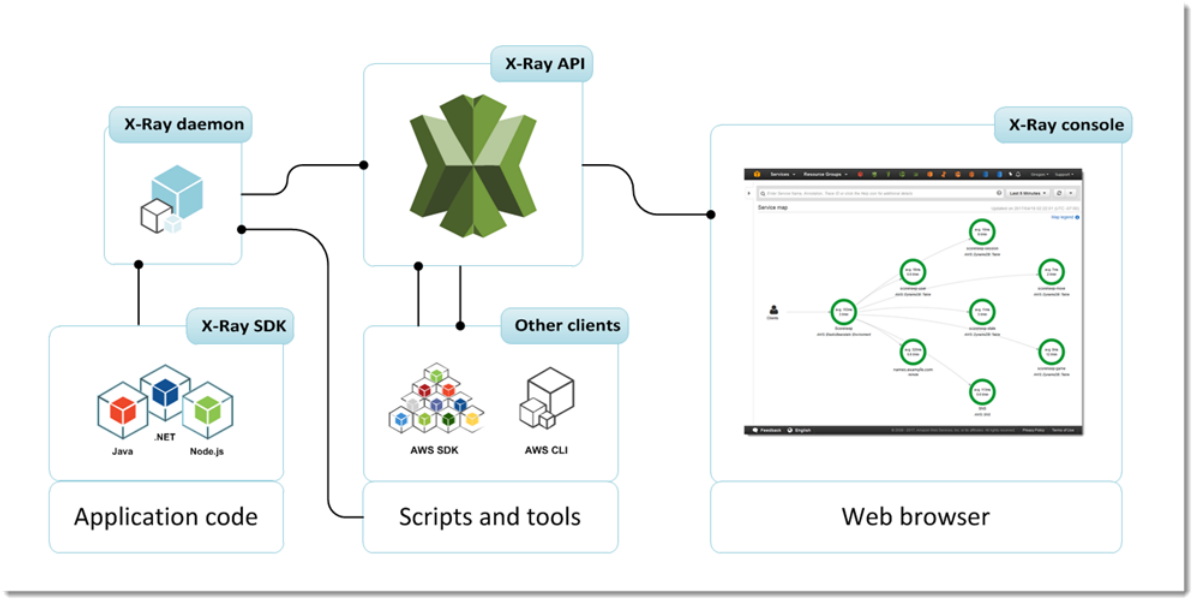
AWS X-Ray is a service that collects data about requests that your application serves, and provides tools you can use to view, filter, and gain insights into that data to identify issues and opportunities for optimization. For any traced request to your application, you can see detailed information not only about the request and response, but also about calls that your application makes to downstream AWS resources, microservices, databases and HTTP web APIs.




The X-Ray SDK provides:

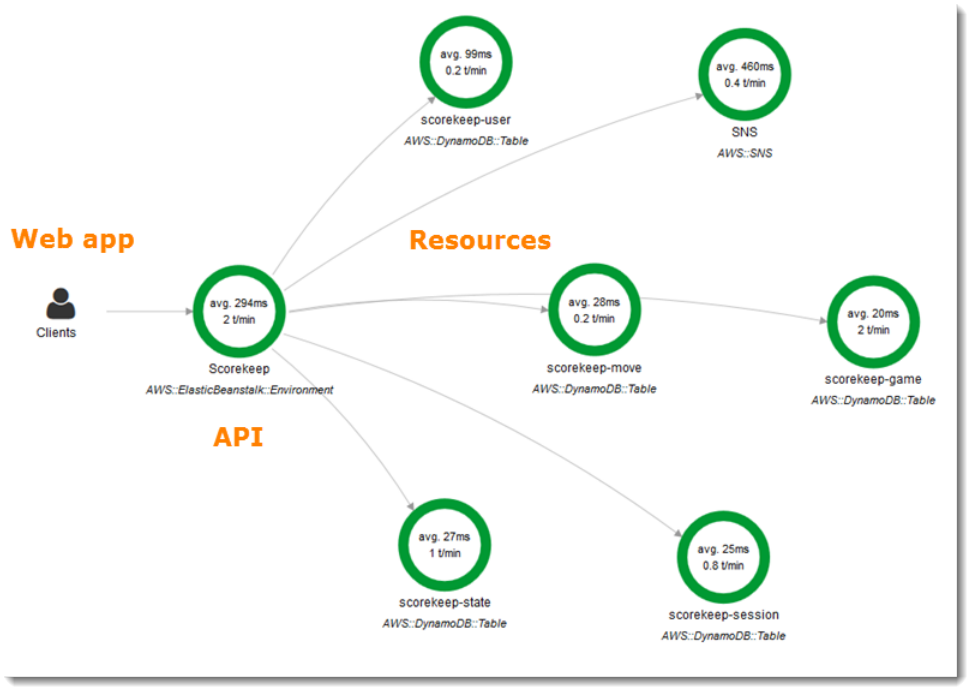
- **Interceptors** to add to your code to trace incoming HTTP requests
- **Client handlers** to instrument AWS SDK clients that your application uses to call other AWS services
- An **HTTP client** to use to instrument calls to other internal and external HTTP web services

The SDK also supports instrumenting calls to SQL databases, automatic AWS SDK client instrumentation, and other features.




Instead of sending trace data directly to X-Ray, the SDK sends JSON segment documents to a daemon process listening for UDP traffic. The X-Ray daemon (p. 137) buffers segments in a queue and uploads them to X-Ray in batches. The daemon is available for Linux, Windows, and macOS, and is included on AWS Elastic Beanstalk and AWS Lambda platforms.

X-Ray uses trace data from the AWS resources that power your cloud applications to generate a detailed **service graph**. The service graph shows the client, your front-end service, and backend services that your front-end service calls to process requests and persist data. Use the service graph to identify bottlenecks, latency spikes, and other issues to solve to improve the performance of your applications.

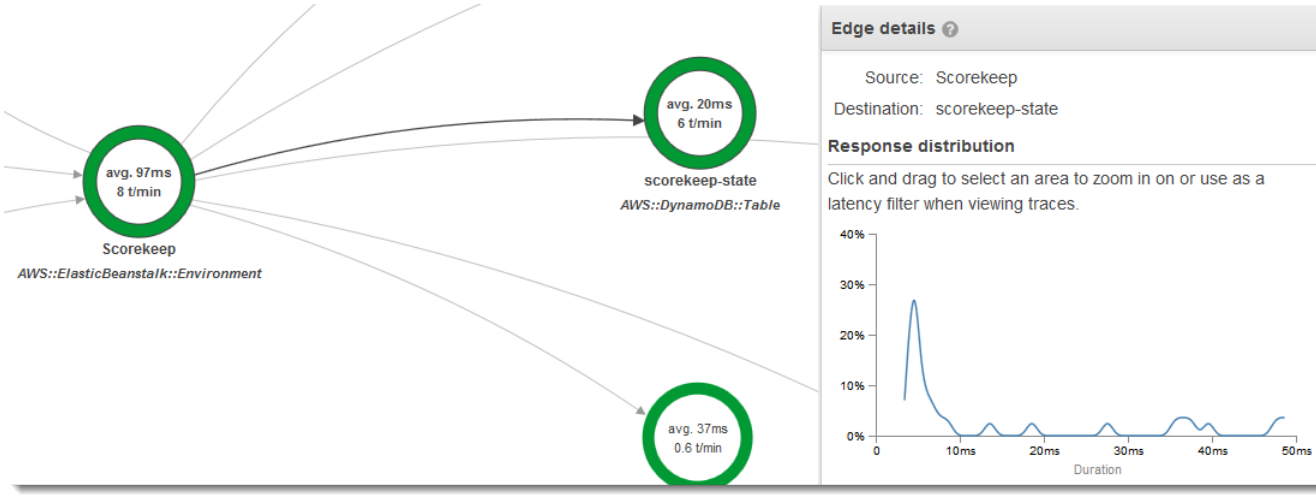



See the getting started tutorial (p. 8) to start using X-Ray in just a few minutes with an instrumented sample application. Or keep reading (p. 3) to learn about the languages, frameworks, and services that work with X-Ray.

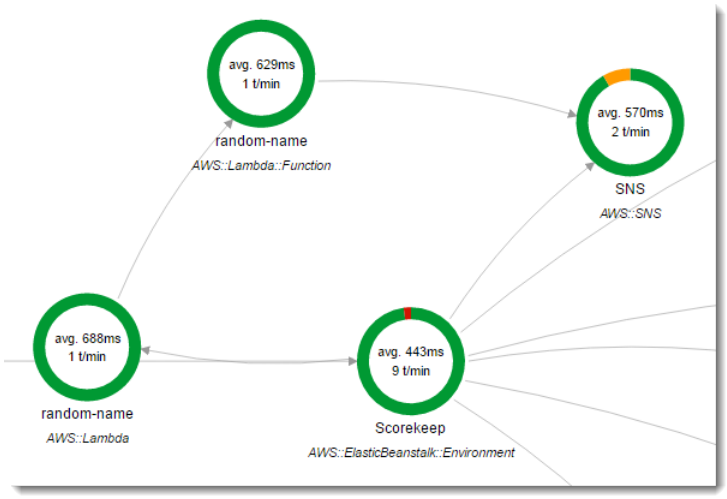
# AWS X-Ray use cases and requirements

You can use the X-Ray SDK and AWS service integration to instrument requests to your applications that are running locally or on AWS compute services such as Amazon EC2, Elastic Beanstalk, Amazon ECS and AWS Lambda.

To instrument your application code, you use the **X-Ray SDK**. The SDK records data about incoming and outgoing requests and sends it to the X-Ray daemon, which relays the data in batches to X-Ray. For example, when your application calls DynamoDB to retrieve user information from a DynamoDB table, the X-Ray SDK records data from both the client request and the downstream call to DynamoDB.




Other AWS services make it easier to instrument your application's components by integrating with X-Ray. **Service integration** can include adding tracing headers to incoming requests, sending trace data to X-Ray, or running the X-Ray daemon. For example, AWS Lambda can send trace data about requests to your Lambda functions, and run the X-Ray daemon on workers to make it easier to use the X-Ray SDK.



Many instrumentation scenarios require only configuration changes. For example, you can instrument all incoming HTTP requests and downstream calls to AWS services that your Java application makes. To do this, you add the X-Ray SDK for Java's filter to your servlet configuration, and take the AWS SDK for Java Instrumentor submodule as a build dependency. For advanced instrumentation, you can modify your application code to customize and annotate the data that the SDK sends to X-Ray.

**Sections**



# Supported languages and frameworks

AWS X-Ray provides tools and integration to support a variety of languages, frameworks, and platforms.

### C#

On Windows Server, you can use the X-Ray SDK for .NET to instrument incoming requests, AWS SDK clients, SQL clients, and HTTP clients. On AWS Lambda, you can use the Lambda X-Ray integration to instrument incoming requests.

See AWS X-Ray SDK for .NET (p. 266) for more information.

- **.NET on Windows Server** – Add a message handler (p. 273) to your HTTP configuration to instrument incoming requests.
- **C# .NET Core on AWS Lambda** – Enable X-Ray on your Lambda function configuration to instrument incoming requests.

### Go

In any Go application, you can use the X-Ray SDK for Go classes to instrument incoming requests, AWS SDK clients, SQL clients, and HTTP clients. Automatic request instrumentation is available for applications that use HTTP handlers.

On AWS Lambda, you can use the Lambda X-Ray integration to instrument incoming requests. Add the X-Ray SDK for Go to your function for full instrumentation.

See AWS X-Ray SDK for Go (p. 174) for more information.

- **Go web applications** – Use the X-Ray SDK for Go HTTP handler (p. 180) to process incoming requests on your routes.
- **Go on AWS Lambda** – Enable X-Ray on your Lambda function configuration to instrument incoming requests. Add the X-Ray SDK for Go to instrument AWS SDK, HTTP, and SQL clients.

### Java

In any Java application, you can use the X-Ray SDK for Java classes to instrument incoming requests, AWS SDK clients, SQL clients, and HTTP clients. Automatic request instrumentation is available for frameworks that support Java servlets. Automatic SDK instrumentation is available through the Instrumentor submodule.

On AWS Lambda, you can use the Lambda X-Ray integration to instrument incoming requests. Add the X-Ray SDK for Java to your function for full instrumentation.

See AWS X-Ray SDK for Java (p. 186) for more information.

- **Tomcat** – Add a servlet filter (p. 197) to your deployment descriptor (`web.xml`) to instrument incoming requests.
- **Spring Boot** – Add a servlet filter (p. 197) to your `WebConfig` class to instrument incoming requests.
- **Java on AWS Lambda** – Enable X-Ray on your Lambda function to instrument incoming requests. Add the X-Ray SDK for Java to instrument AWS SDK, HTTP, and SQL clients.
- **Other frameworks** – Add a servlet filter if your framework supports servlets, or manually create a segment for each incoming request.

### Node.js

In any Node.js application, you can use the X-Ray SDK for Node.js classes to instrument incoming requests, AWS SDK clients, SQL clients, and HTTP clients. Automatic request instrumentation is available for applications that use the Express and Restify frameworks.

On AWS Lambda, you can use the Lambda X-Ray integration to instrument incoming requests. Add the X-Ray SDK for Node.js to your function for full instrumentation.

See The X-Ray SDK for Node.js (p. 214) for more information.

- **Express or Restify** – Use the X-Ray SDK for Node.js middleware (p. 219) to instrument incoming requests.
- **Node.js on AWS Lambda** – Enable X-Ray on your Lambda function to instrument incoming requests. Add the X-Ray SDK for Node.js to instrument AWS SDK, HTTP, and SQL clients
- **Other frameworks** – Manually create a segment for each incoming request.

### Python

In any Python application, you can use the X-Ray SDK for Python classes to instrument incoming requests, AWS SDK clients, SQL clients, and HTTP clients. Automatic request instrumentation is available for applications that use the Django and Flask frameworks.

On AWS Lambda, you can use the Lambda X-Ray integration to instrument incoming requests. Add the X-Ray SDK for Python to your function for full instrumentation.

See AWS X-Ray SDK for Python (p. 230) for more information.

- **Django or Flask** – Use the X-Ray SDK for Python middleware (p. 237) to instrument incoming requests.
- **Python on AWS Lambda** – Enable X-Ray on your Lambda function configuration to instrument incoming requests. Add the X-Ray SDK for Python to instrument AWS SDK, HTTP, and SQL clients.
- **Other frameworks** – Manually create a segment for each incoming request.

### Ruby

In any Ruby application, you can use the X-Ray SDK for Ruby classes to instrument incoming requests, AWS SDK clients, SQL clients, and HTTP clients. Automatic request instrumentation is available for applications that use the Rails framework.

- **Rails** – Add the X-Ray SDK for Ruby gem and railtie to your gemfile, and configure the recorder (p. 259) in an initializer to instrument incoming requests.
- **Other frameworks** – Manually create a segment (p. 260) for each incoming request.

If the X-Ray SDK isn't available for your language or platform, you can generate trace data manually and send it to the X-Ray daemon, or directly to the X-Ray API (p. 79).

# Supported AWS services

Several AWS services provide **X-Ray integration**. Integrated services (p. 153) offer varying levels of integration that can include sampling and adding headers to incoming requests, running the X-Ray daemon, and automatically sending trace data to X-Ray.

- **Active instrumentation** – Samples and instruments incoming requests.
- **Passive instrumentation** – Instruments requests that have been sampled by another service.
- **Request tracing** – Adds a tracing header to all incoming requests and propagates it downstream.
- **Tooling** – Runs the X-Ray daemon to receive segments from the X-Ray SDK.

The following services provide X-Ray integration:

- **AWS Lambda** – Active and passive instrumentation of incoming requests on all runtimes. AWS Lambda adds two nodes to your service map, one for the AWS Lambda service, and one for the function. When you enable instrumentation, AWS Lambda also runs the X-Ray daemon on Java and Node.js runtimes for use with the X-Ray SDK. Learn more (p. 167).
- **Amazon API Gateway** – Active and passive instrumentation. API Gateway uses sampling rules to determine which requests to record, and adds a node for the gateway stage to your service map. Learn more (p. 153).
- **Elastic Load Balancing** – Request tracing on application load balancers. The application load balancer adds the trace ID to the request header before sending it to a target group. Learn more (p. 166).
- **AWS Elastic Beanstalk** – Tooling. Elastic Beanstalk includes the X-Ray daemon on the following platforms:
  - **Java SE** – 2.3.0 and later configurations
  - **Tomcat** – 2.4.0 and later configurations
  - **Node.js** – 3.2.0 and later configurations
  - **Windows Server** – All configurations other than Windows Server Core that have been released since December 9th, 2016.

  You can use the Elastic Beanstalk console to tell Elastic Beanstalk to run the daemon on these platforms, or use the `XRayEnabled` option in the `aws:elasticbeanstalk:xray` namespace. Learn more (p. 166).
- **Amazon Simple Notification Service** – Passive instrumentation. If an Amazon SNS publisher traces its client with the X-Ray SDK, subscribers can retrieve the tracing header and continue to propagate the original trace from the publisher with the same trace ID. Learn more (p. 167).
- **Amazon Simple Queue Service** – Passive instrumentation. If a service traces requests by using the X-Ray SDK, Amazon SQS can send the tracing header and continue to propagate the original trace from the sender to the consumer with a consistent trace ID. Learn more (p. 171).

# Code and configuration changes

A large amount of tracing data can be generated without any functional changes to your code. Detailed tracing of front-end and downstream calls requires only minimal changes to build and deploy-time configuration.

### Examples of Code and Configuration Changes

- **AWS resource configuration** – Change AWS resource settings to instrument requests to a Lambda function. Run the X-Ray daemon on the instances in your Elastic Beanstalk environment by changing an option setting.

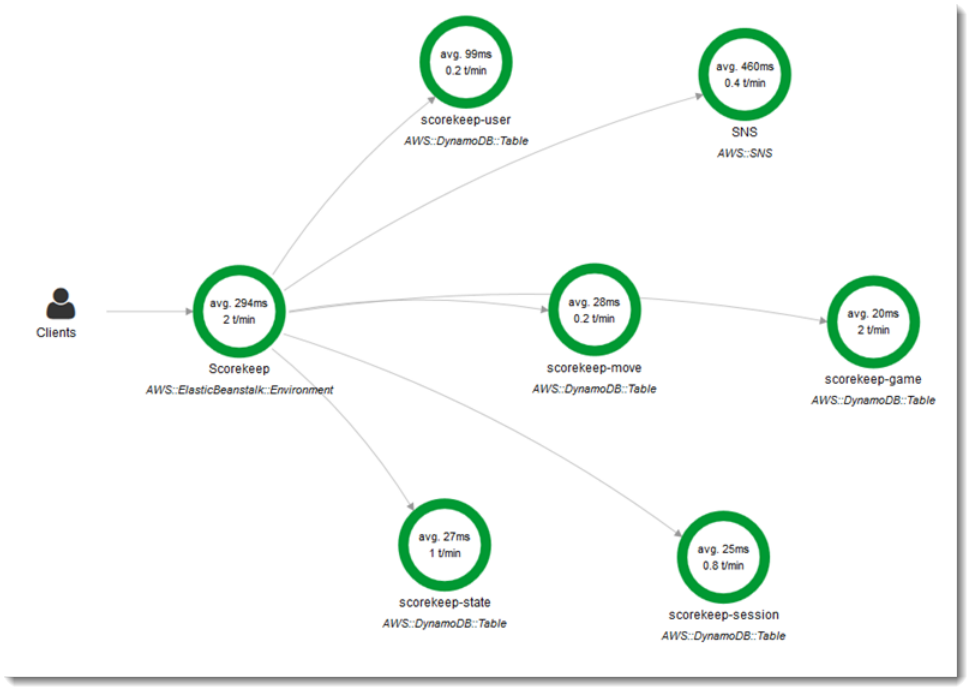
- **Build configuration** – Take X-Ray SDK for Java submodules as a compile-time dependency to instrument all downstream requests to AWS services, and to resources such as Amazon DynamoDB tables, Amazon SQS queues, and Amazon S3 buckets.
- **Application configuration** – To instrument incoming HTTP requests, add a servlet filter to your Java application, or use the X-Ray SDK for Node.js as middleware on your Express application. Change sampling rules and enable plugins to instrument the Amazon EC2, Amazon ECS, and AWS Elastic Beanstalk resources that run your application.
- **Class or object configuration** – To instrument outgoing HTTP calls in Java, import the X-Ray SDK for Java version of `HttpClientBuilder` instead of the Apache.org version.
- **Functional changes** – Add a request handler to an AWS SDK client to instrument calls that it makes to AWS services. Create subsegments to group downstream calls, and add debug information to segments with annotations and metadata.

# Getting started with AWS X-Ray

To get started with AWS X-Ray, launch a sample app in Elastic Beanstalk that is already instrumented (p. 186) to generate trace data. In a few minutes, you can launch the sample app, generate traffic, send segments to X-Ray, and view a service graph and traces in the AWS Management Console.

This tutorial uses a sample Java application (p. 115) to generate segments and send them to X-Ray. The application uses the Spring framework to implement a JSON web API and the AWS SDK for Java to persist data to Amazon DynamoDB. A servlet filter in the application instruments all incoming requests served by the application, and a request handler on the AWS SDK client instruments downstream calls to DynamoDB.


You use the X-Ray console to view the connections among client, server, and DynamoDB in a service map. The service map is a visual representation of the services that make up your web application, generated from the trace data that it generates by serving requests.




With the X-Ray SDK for Java, you can trace all of your application's primary and downstream AWS resources by making two configuration changes:

- Add the X-Ray SDK for Java's tracing filter to your servlet configuration in a `WebConfig` class or `web.xml` file.
- Take the X-Ray SDK for Java's submodules as build dependencies in your Maven or Gradle build configuration.

You can also access the raw service map and trace data by using the AWS CLI to call the X-Ray API. The service map and trace data are JSON that you can query to ensure that your application is sending data, or to check specific fields as part of your test automation.

**Sections**



# Prerequisites

This tutorial uses Elastic Beanstalk to create and configure the resources that run the sample application and X-Ray daemon. If you use an IAM user with limited permissions, add the Elastic Beanstalk managed user policy to grant your IAM user permission to use Elastic Beanstalk, and the

`AWSXrayReadOnlyAccess` managed policy for permission to read the service map and traces in the X-Ray console.

Create an Elastic Beanstalk environment for the sample application. If you haven't used Elastic Beanstalk before, this will also create a service role and instance profile for your application. A default VPC must exist in the region you're going to deploy to or Elastic Beanstalk will fail to deploy the sample application.

**To create an Elastic Beanstalk environment**

1. Open the Elastic Beanstalk Management Console with this preconfigured link: https://console.aws.amazon.com/elasticbeanstalk/#/newApplication?applicationName=scorekeep&solutionStackName=Java
2. Choose **Create application** to create an application with an environment running the Java 8 SE platform.
3. When your environment is ready, the console redirects you to the environment Dashboard.
4. Click the URL at the top of the page to open the site.

The instances in your environment need permission to send data to the AWS X-Ray service. Additionally, the sample application uses Amazon S3 and DynamoDB. Modify the default Elastic Beanstalk instance profile to include permissions to use these services.

**To add X-Ray, Amazon S3 and DynamoDB permissions to your Elastic Beanstalk environment**

1. Open the Elastic Beanstalk instance profile in the IAM console: aws-elasticbeanstalk-ec2-role.
2. Choose **Attach Policies**.
3. Attach **AWSXrayFullAccess**, **AmazonS3FullAccess**, and **AmazonDynamoDBFullAccess** to the role.

   **Note**
   Full access policies are not best practice for general use. For instructions on configuring a policy with least privilege to reduce security risks, see Optional: Least privilege policy (p. 18).

# Deploy to Elastic Beanstalk and generate trace data

Deploy the sample application to your Elastic Beanstalk environment. The sample application uses Elastic Beanstalk configuration files to configure the environment for use with X-Ray and create the DynamoDB that it uses automatically.

**To deploy the source code**

1. Download the sample app: eb-java-scorekeep-xray-gettingstarted-v1.3.zip
2. Open the Elastic Beanstalk console.
3. Navigate to the management console for your environment.
4. Choose **Upload and Deploy**.
5. Upload eb-java-scorekeep-xray-gettingstarted-v1.3.zip, and then choose **Deploy**.

The sample application includes a front-end web app. Use the web app to generate traffic to the API and send trace data to X-Ray.

### To generate trace data

1. In the environment Dashboard, click the URL to open the web app.
2. Choose **Create** to create a user and session.
3. Type a **game name**, set the **Rules** to **Tic Tac Toe**, and then choose **Create** to create a game.
4. Choose **Play** to start the game.
5. Choose a tile to make a move and change the game state.

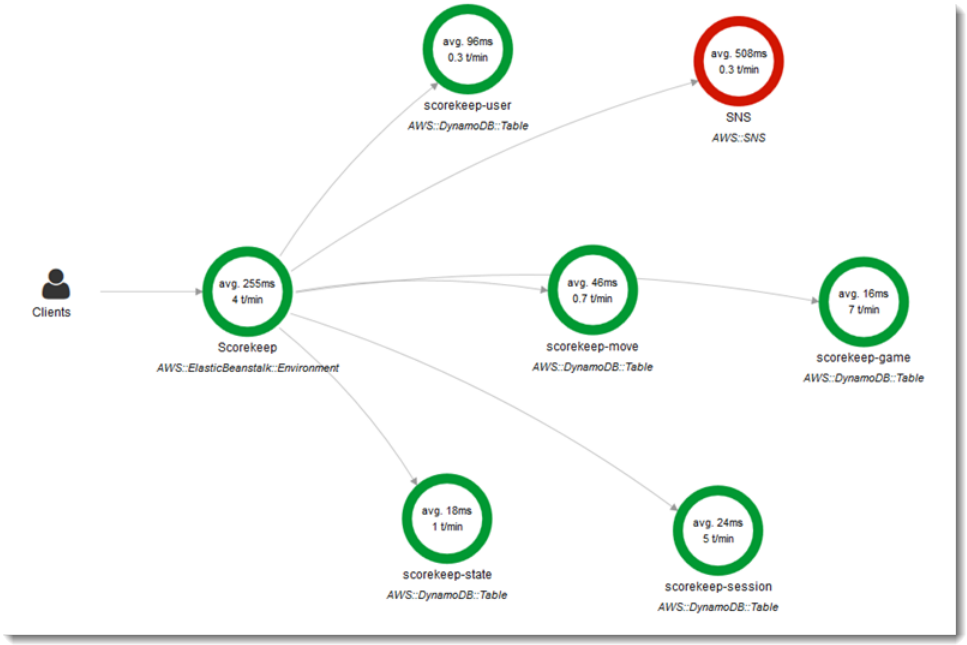
Each of these steps generates HTTP requests to the API, and downstream calls to DynamoDB to read and write user, session, game, move, and state data.

# View the service map in the X-Ray console

You can see the service map and traces generated by the sample application in the X-Ray console.

### To use the X-Ray console

1. Open the service map page of the X-Ray console.
2. The console shows a representation of the service graph that X-Ray generates from the trace data sent by the application.

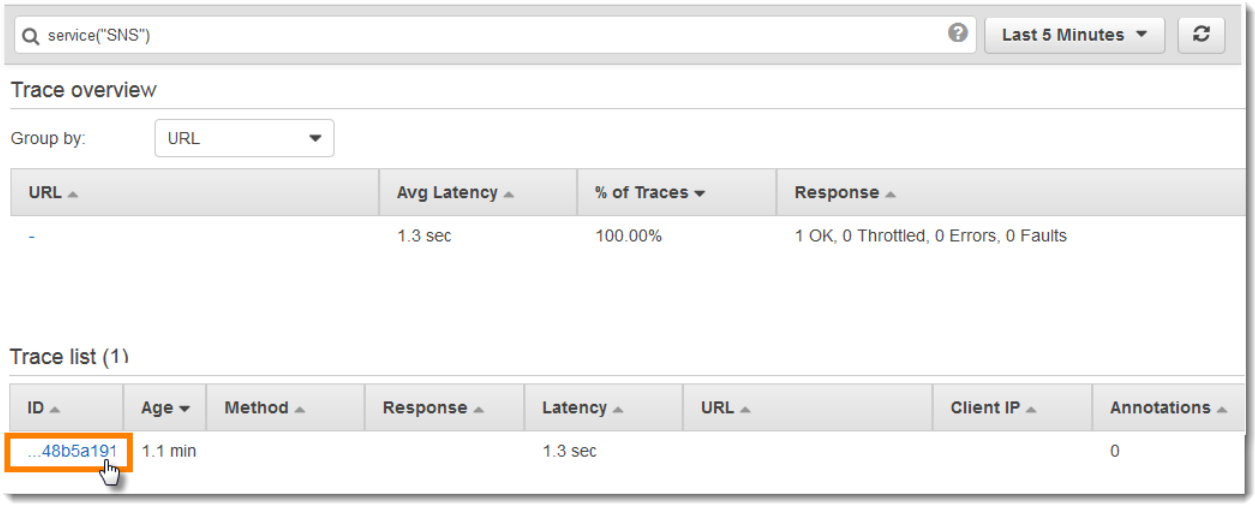



The service map shows the web app client, the API running in Elastic Beanstalk, the DynamoDB service, and each DynamoDB table that the application uses. Every request to the application, up to a configurable maximum number of requests per second, is traced as it hits the API, generates requests to downstream services, and completes.

You can choose any node in the service graph to view traces for requests that generated traffic to that node. Currently, the Amazon SNS node is red. Drill down to find out why.

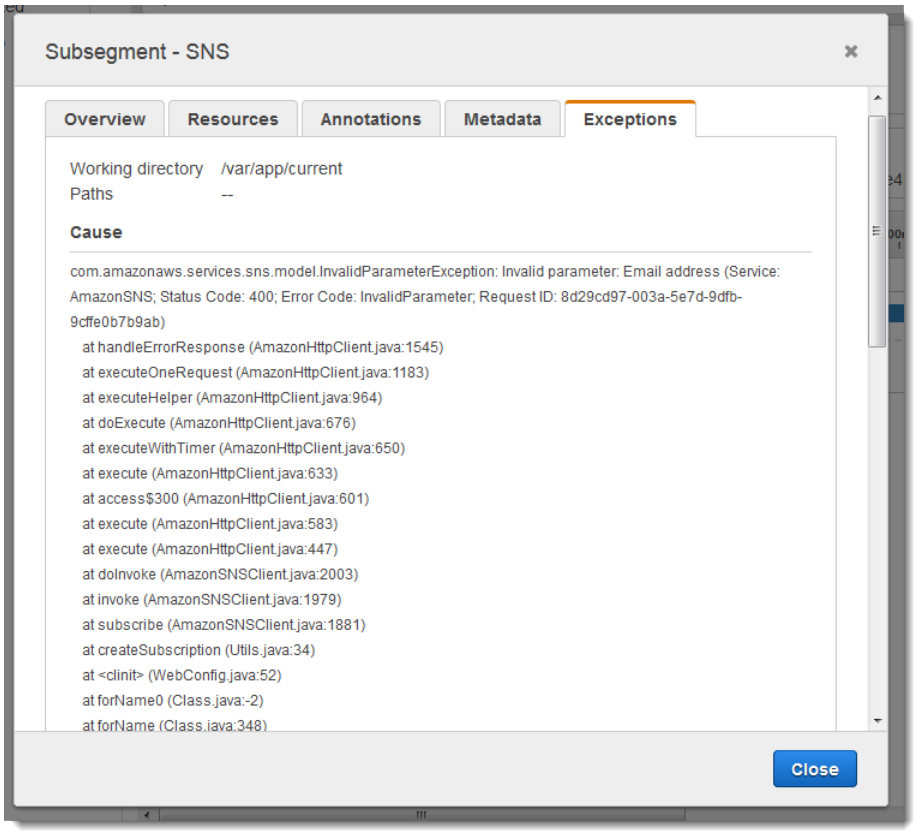
### To find the cause of the error

1. Choose the node named **SNS**. The **Service details panel** opens to the right.
2. Choose **View traces** to access the **Trace overview** screen.
3. Choose the trace from the **Trace list**. This trace doesn't have a method or URL because it was recorded during startup instead of in response to an incoming request.




4. Choose the red status icon to open the **Exceptions** page for the SNS subsegment.


5. The X-Ray SDK automatically captures exceptions thrown by instrumented AWS SDK clients and records the stack trace.



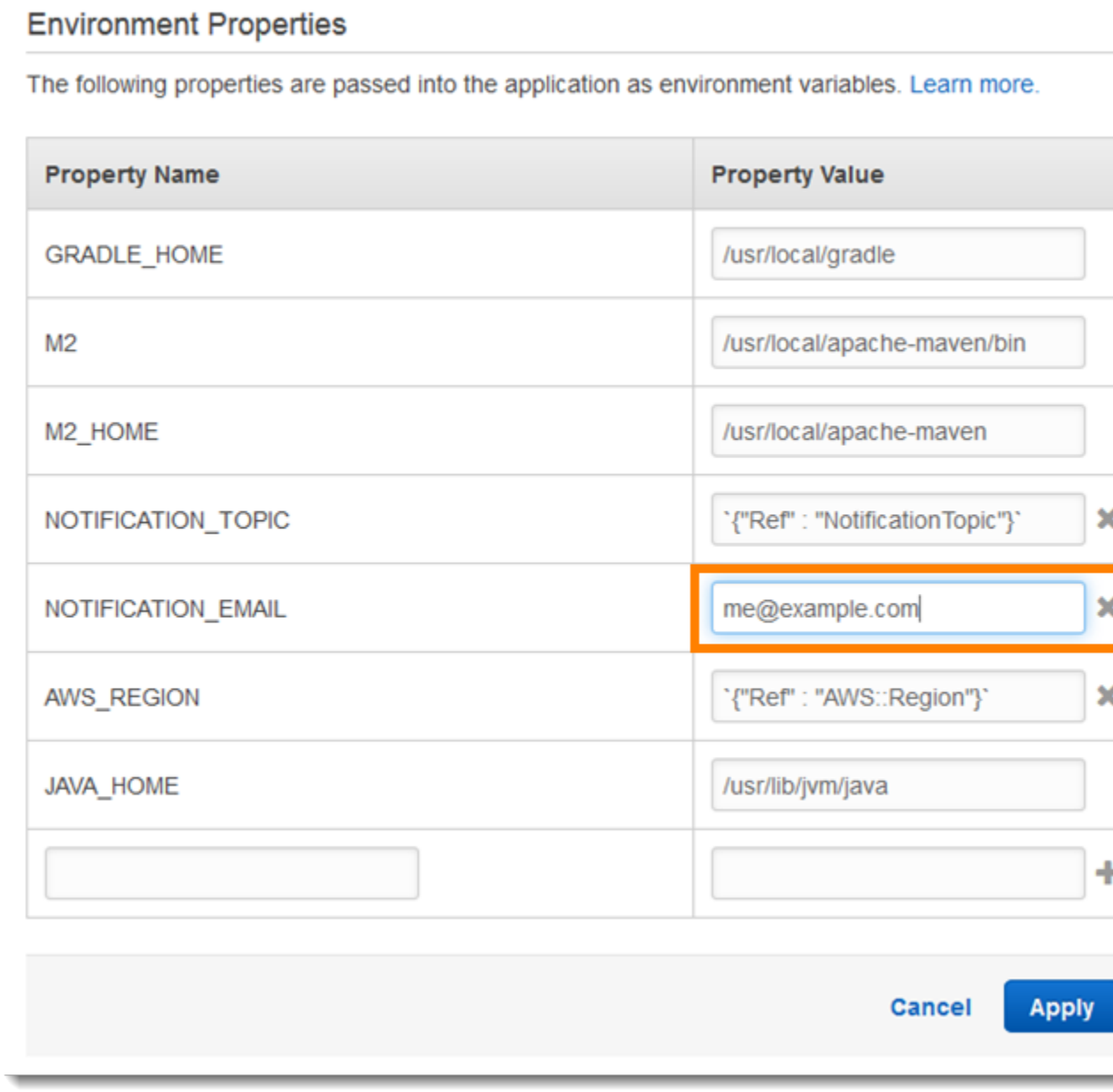

The cause indicates that the email address provided in a call to `createSubscription` made in the `WebConfig` class was invalid. Let's fix that.

# Configuring Amazon SNS notifications

Scorekeep uses Amazon SNS to send notifications when users complete a game. When the application starts up, it tries to create a subscription for an email address defined in an environment variable. That call is currently failing, causing the Amazon SNS node in your service map to be red. Configure a notification email in an environment variable to enable notifications and make the service map green.

**To configure Amazon SNS notifications for scorekeep**

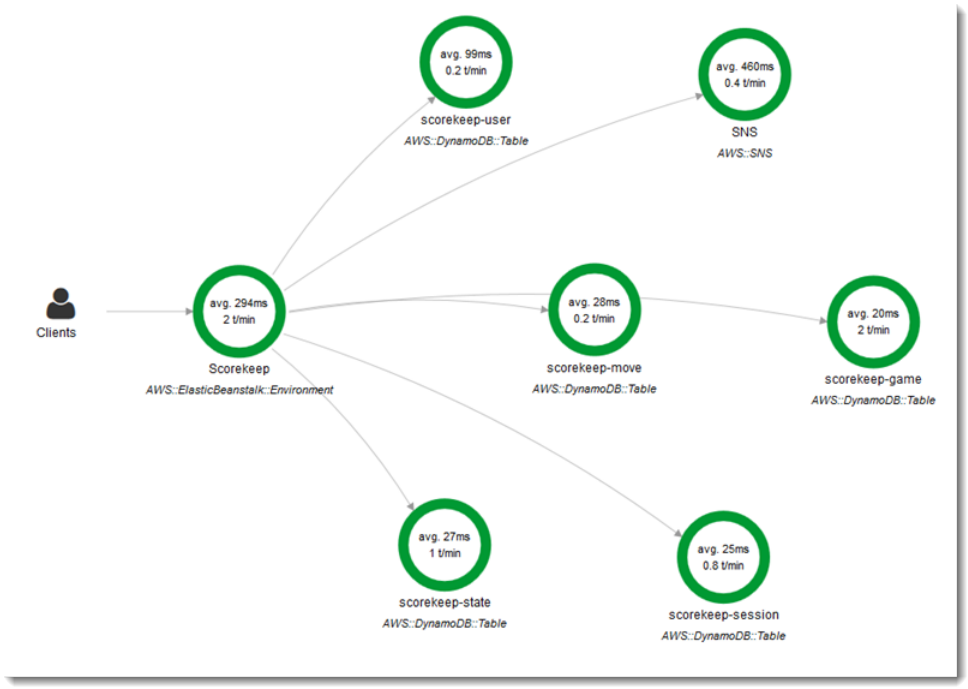
1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Choose **Configuration**.
4. Choose **Software Configuration**.
5. Under **Environment Properties**, replace the default value with your email address.




**Note**
The default value uses an AWS CloudFormation function to retrieve a parameter stored in a configuration file (a dummy value in this case).

6. Choose **Apply**.

When the update completes, Scorekeep restarts and creates a subscription to the SNS topic. Check your email and confirm the subscription to see updates when you complete a game.




# Explore the sample application

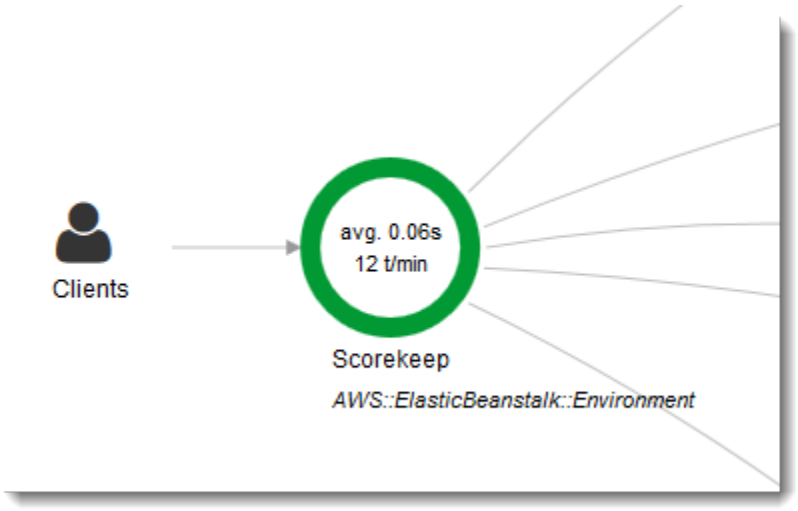
The sample application is an HTTP web API in Java that is configured to use the X-Ray SDK for Java. When you deploy the application to Elastic Beanstalk, it creates the DynamoDB tables, compiles the API with Gradle, and configures the nginx proxy server to serve the web app statically at the root path. At the same time, Elastic Beanstalk routes requests to paths starting with `/api` to the API.

To instrument incoming HTTP requests, the application adds the `TracingFilter` provided by the SDK.

**Example src/main/java/scorekeep/WebConfig.java - servlet filter**

```
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
...

@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
...
```

This filter sends trace data about all incoming requests that the application serves, including request URL, method, response status, start time, and end time.




The application also makes downstream calls to DynamoDB using the AWS SDK for Java. To instrument these calls, the application simply takes the AWS SDK-related submodules as dependencies, and the X-Ray SDK for Java automatically instruments all AWS SDK clients.

The application uses a `Buildfile` file to build the source code on-instance with `Gradle` and a `Procfile` file to run the executable JAR that Gradle generates. `Buildfile` and `Procfile` support is a feature of the Elastic Beanstalk Java SE platform.

**Example Buildfile**

```
build: gradle build
```

**Example Procfile**

```
web: java -Dserver.port=5000 -jar build/libs/scorekeep-api-1.0.0.jar
```

The `build.gradle` file downloads the SDK submodules from Maven during compilation by declaring them as dependencies.

**Example build.gradle -- dependencies**

```
...
dependencies {
    compile("org.springframework.boot:spring-boot-starter-web")
    testCompile('org.springframework.boot:spring-boot-starter-test')
    compile('com.amazonaws:aws-java-sdk-dynamodb')
    compile("com.amazonaws:aws-xray-recorder-sdk-core")
    compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk")
    compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-instrumentor")
    ...
}
dependencyManagement {
    imports {
        mavenBom("com.amazonaws:aws-java-sdk-bom:1.11.67")
        mavenBom("com.amazonaws:aws-xray-recorder-sdk-bom:2.4.0")
    }
}
```

The core, AWS SDK, and AWS SDK Instrumentor submodules are all that's required to automatically instrument any downstream calls made with the AWS SDK.

To run the X-Ray daemon, the application uses another feature of Elastic Beanstalk, configuration files. The configuration file tells Elastic Beanstalk to run the daemon and send its log on demand.

**Example .ebextensions/xray.config**

```
option_settings:
  aws:elasticbeanstalk:xray:
    XRayEnabled: true

files:
  "/opt/elasticbeanstalk/tasks/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      /var/log/xray/xray.log
```

The X-Ray SDK for Java provides a class named `AWSXRay` that provides the global recorder, a `TracingHandler` that you can use to instrument your code. You can configure the global recorder to customize the `AWSXRayServletFilter` that creates segments for incoming HTTP calls. The sample includes a static block in the `WebConfig` class that configures the global recorder with plugins and sampling rules.

**Example src/main/java/scorekeep/WebConfig.java - recorder**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {
...
  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin()).withPlugin(new ElasticBeanstalkPlugin());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
  }
}
```

This example uses the builder to load sampling rules from a file named `sampling-rules.json`. Sampling rules determine the rate at which the SDK records segments for incoming requests.

**Example src/main/java/resources/sampling-rules.json**

```
{
  "version": 1,
  "rules": [
    {
      "description": "Resource creation.",
      "service_name": "*",
      "http_method": "POST",
      "url_path": "/api/*",
      "fixed_target": 1,
      "rate": 1.0
    },
    {
```

```
      "description": "Session polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/session/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    {
      "description": "Game polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/game/*/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    {
      "description": "State polling.",
      "service_name": "*",
      "http_method": "GET",
      "url_path": "/api/state/*/*/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

The sampling rules file defines four custom sampling rules and the default rule. For each incoming request, the SDK evaluates the custom rules in the order in which they are defined. The SDK applies the first rule that matches the request's method, path, and service name. For Scorekeep, the first rule catches all POST requests (resource creation calls) by applying a fixed target of one request per second and a rate of 1.0, or 100 percent of requests after the fixed target is satisfied.

The other three custom rules apply a five percent rate with no fixed target to session, game, and state reads (GET requests). This minimizes the number of traces for periodic calls that the front end makes automatically every few seconds to ensure the content is up to date. For all other requests, the file defines a default rate of one request per second and a rate of 10 percent.

The sample application also shows how to use advanced features such as manual SDK client instrumentation, creating additional subsegments, and outgoing HTTP calls. For more information, see AWS X-Ray sample application (p. 115).

# Optional: Least privilege policy

You have just deployed this tutorial using the **AmazonS3FullAccess** and **AmazonDynamoDBFullAccess** security policies. Using a full access policy isn't the best practice in the long term. To improve the security of what you deployed, follow these steps to update your permissions. To learn more about security best practices in IAM policies, see Identity and access management for AWS X-Ray.

To update your policies, first you identify the ARNs of your Amazon S3 and DynamoDB resources. Then you use the ARNs in two custom IAM policies. Finally, you apply those policies to your instance profile.

### To identify your Amazon S3 resource

1. Open the Resources page of the AWS Config console.
2. Under **Resource type**, filter by **AWS S3 Bucket** to find the ARN of the Amazon S3 bucket that your application uses.

3. Choose the **Resource identifier** that's attached to `elasticbeanstalk`.
4. Record its full **Amazon resource name**.
5. Insert the ARN into the following IAM policy.

**Example**

```
{
   "Version": "2012-10-17",
   "Statement": [
   {
      "Sid": "ScorekeepS3",
      "Action": [
         "s3:GetObject",
         "s3:PutObject"
      ],
      "Effect": "Allow",
      "Resource": "arn:aws:s3:::elasticbeanstalk-region-0987654321"
   }
   ]
}
```

### To identify your DynamoDB resource

1. Open the Resources page of the AWS Config console.
2. Under **Resource type**, filter by **AWS DynamoDB Table** to find the ARN of the DynamoDB tables that your application uses.
3. Choose the **Resource identifier** that's attached to one of the `scorekeep` tables.
4. Record its full **Amazon resource name**.
5. Insert the ARN into the following IAM policy.

**Example**

```
{
   "Version": "2012-10-17",
   "Statement": [
   {
      "Sid": "ScorekeepDynamoDB",
      "Effect": "Allow",
      "Action": [
         "dynamodb:PutItem",
         "dynamodb:UpdateItem",
         "dynamodb:DeleteItem",
         "dynamodb:GetItem"
      ],
      "Resource": "arn:aws:dynamodb:region:1234567890:table/scorekeep-*"
   }
   ]
}
```

The tables that the application creates follow a consistent naming convention. You can use the format of `scorekeep-*` to indicate all tables following that convention.

### To change your IAM policy

1. Open the Elastic Beanstalk instance profile in the IAM console: aws-elasticbeanstalk-ec2-role.
2. Remove the **AmazonS3FullAccess** and **AmazonDynamoDBFullAccess** policies from the role.

3. Choose **Attach policies**, and then **Create policy**.
4. Choose the **JSON** and paste in one of the policies created previously.
5. Choose **Review policy**.
6. For **Name**, assign a name.
7. Choose **Create policy**.
8. Assign the newly created policy to the `aws-elasticbeanstalk-ec2-role`.
9. Repeat for the second policy created previously.

# Clean up

Terminate your Elastic Beanstalk environment to shut down the Amazon EC2 instances, DynamoDB tables, and other resources.

**To terminate your Elastic Beanstalk environment**

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Choose **Actions**.
4. Choose **Terminate Environment**.
5. Choose **Terminate**.

Trace data is automatically deleted from X-Ray after 30 days.

# Next steps

Learn more about X-Ray in the next chapter, AWS X-Ray concepts (p. 21).

To instrument your own app, learn more about the X-Ray SDK for Java or one of the other X-Ray SDKs:

- **X-Ray SDK for Java** – AWS X-Ray SDK for Java (p. 186)
- **X-Ray SDK for Node.js** – The X-Ray SDK for Node.js (p. 214)
- **X-Ray SDK for .NET** – AWS X-Ray SDK for .NET (p. 266)

To run the X-Ray daemon locally or on AWS, see AWS X-Ray daemon (p. 137).

To contribute to the sample application on GitHub, see eb-java-scorekeep.

# AWS X-Ray concepts

AWS X-Ray receives data from services as *segments*. X-Ray then groups segments that have a common request into *traces*. X-Ray processes the traces to generate a *service graph* that provides a visual representation of your application.

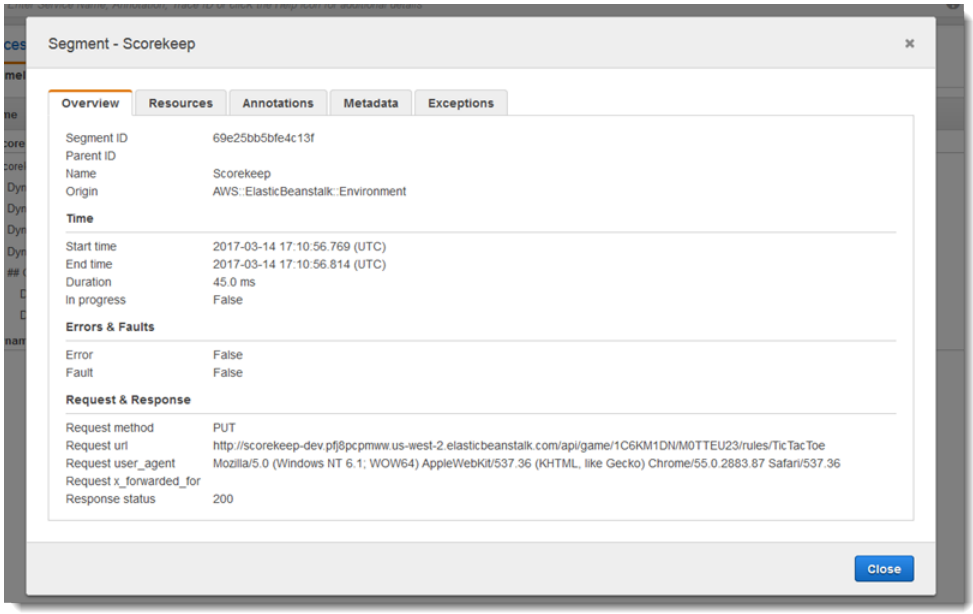
**Concepts**



## Segments

The compute resources running your application logic send data about their work as **segments**. A segment provides the resource's name, details about the request, and details about the work done. For example, when an HTTP request reaches your application, it can record the following data about:

- **The host** – hostname, alias or IP address
- **The request** – method, client address, path, user agent
- **The response** – status, content
- **The work done** – start and end times, subsegments
- **Issues that occur** – errors, faults and exceptions (p. 28), including automatic capture of exception stacks.



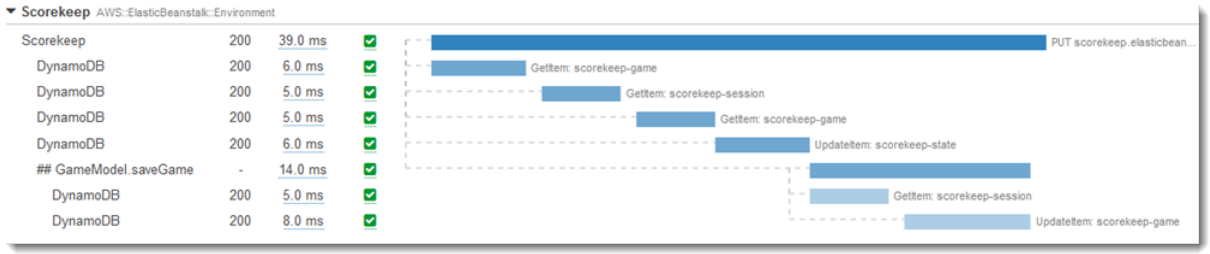

The X-Ray SDK gathers information from request and response headers, the code in your application, and metadata about the AWS resources on which it runs. You choose the data to collect by modifying your application configuration or code to instrument incoming requests, downstream requests, and AWS SDK clients.

**Forwarded Requests**
If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

You can use the X-Ray SDK to record additional information such as annotations and metadata. (p. 27) For details about the structure and information that is recorded in segments and subsegments, see AWS X-Ray segment documents (p. 103). Segment documents can be up to 64 kB in size.

# Subsegments

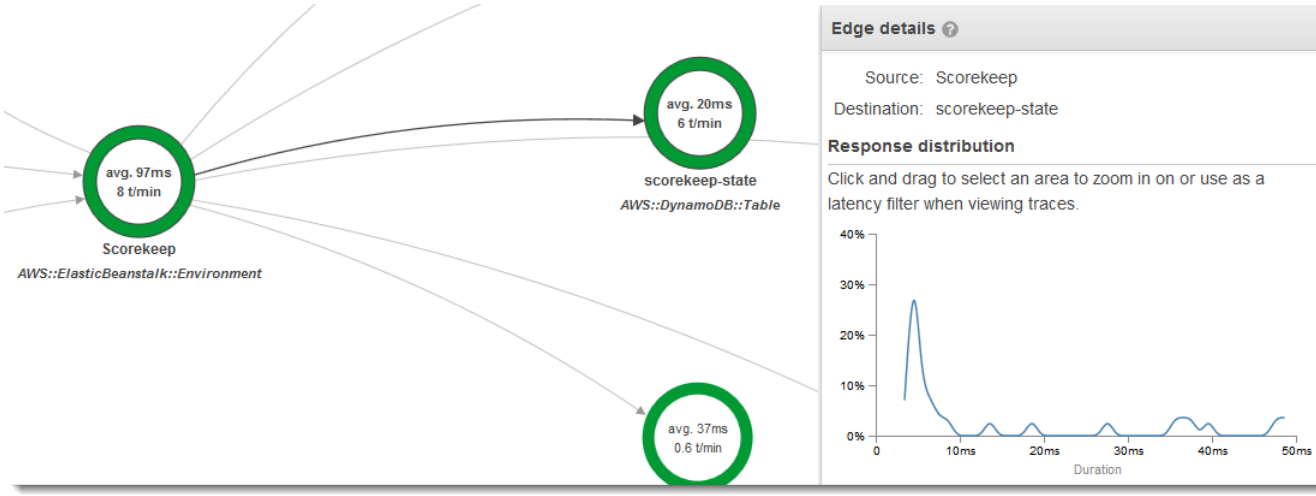
A segment can break down the data about the work done into **subsegments**. Subsegments provide more granular timing information and details about downstream calls that your application made to fulfill the original request. A subsegment can contain additional details about a call to an AWS service, an external HTTP API, or an SQL database. You can even define arbitrary subsegments to instrument specific functions or lines of code in your application.



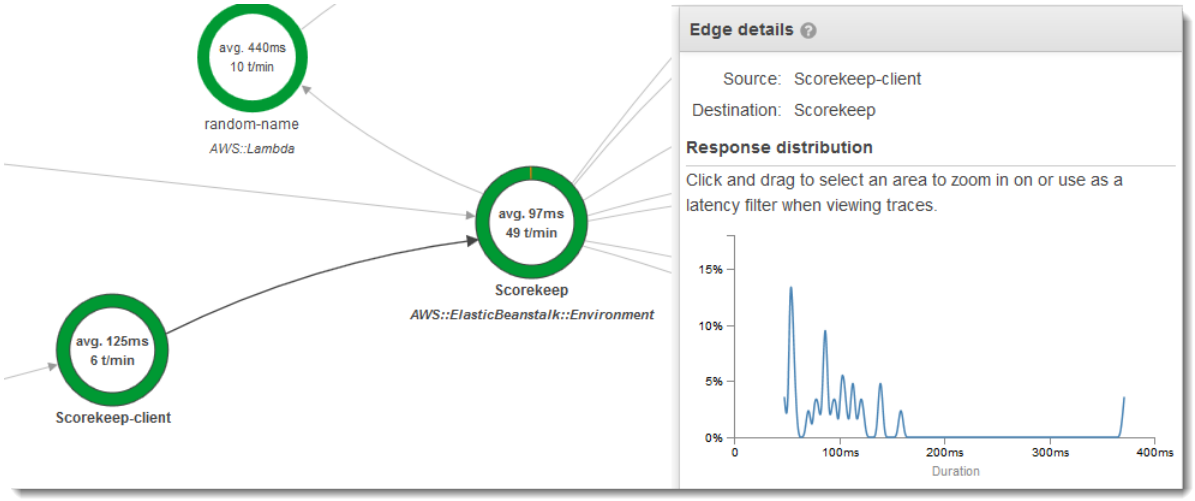
For services that don't send their own segments, like Amazon DynamoDB, X-Ray uses subsegments to generate *inferred segments* and downstream nodes on the service map. This lets you see all of your downstream dependencies, even if they don't support tracing, or are external.

Subsegments represent your application's view of a downstream call as a client. If the downstream service is also instrumented, the segment that it sends replaces the inferred segment generated from the upstream client's subsegment. The node on the service graph always uses information from the service's segment, if it's available, while the edge between the two nodes uses the upstream service's subsegment.

For example, when you call DynamoDB with an instrumented AWS SDK client, the X-Ray SDK records a subsegment for that call. DynamoDB doesn't send a segment, so the inferred segment in the trace, the DynamoDB node on the service graph, and the edge between your service and DynamoDB all contain information from the subsegment.




When you call another instrumented service with an instrumented application, the downstream service sends its own segment to record its view of the same call that the upstream service recorded in a subsegment. In the service graph, both services' nodes contain timing and error information from those services' segments, while the edge between them contains information from the upstream service's subsegment.



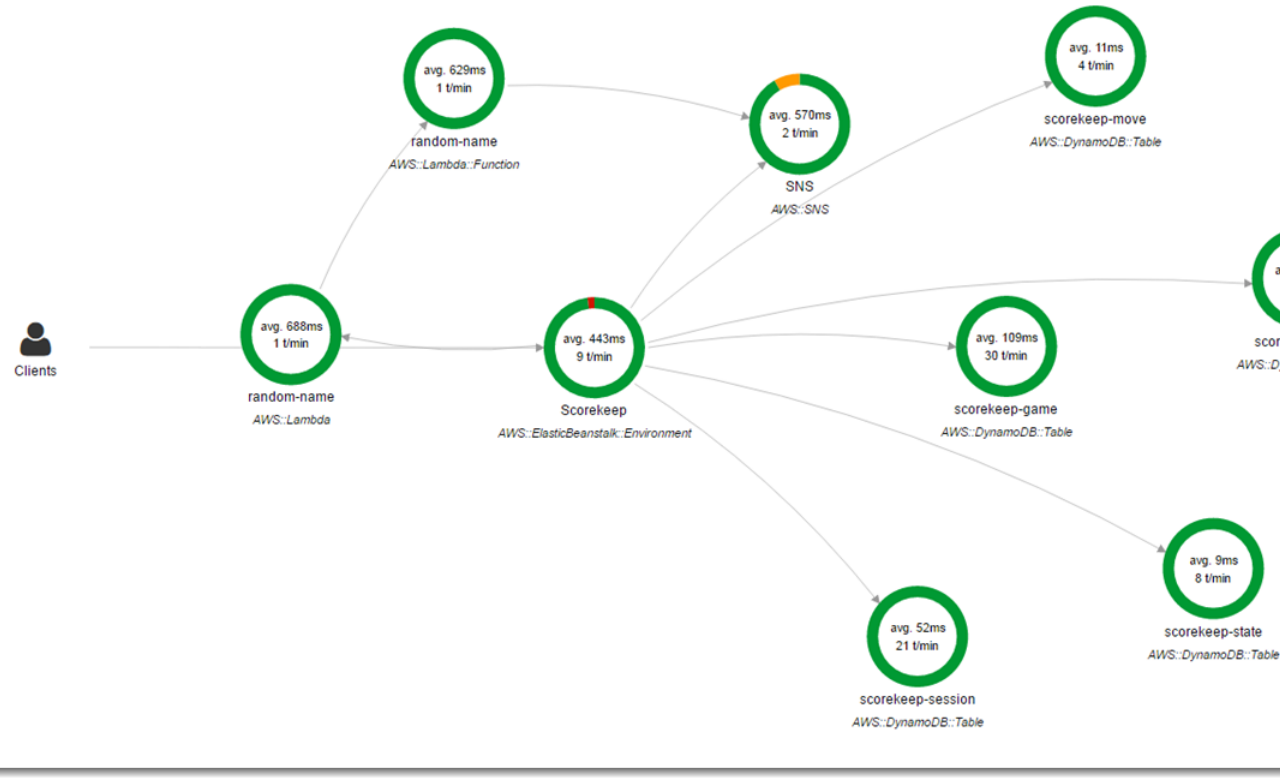

Both viewpoints are useful, as the downstream service records precisely when it started and ended work on the request, and the upstream service records the round trip latency, including time that the request spent traveling between the two services.

# Service graph

X-Ray uses the data that your application sends to generate a **service graph**. Each AWS resource that sends data to X-Ray appears as a service in the graph. **Edges** connect the services that work together to serve requests. Edges connect clients to your application, and your application to the downstream services and resources that it uses.

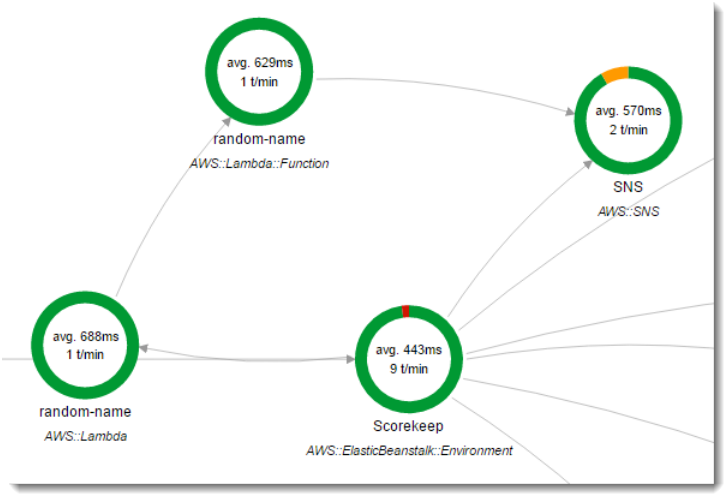
> **Service Names**
> A segment's `name` should match the domain name or logical name of the service that generates the segment. However, this is not enforced. Any application that has permission to `PutTraceSegments` can send segments with any name.

A service graph is a JSON document that contains information about the services and resources that make up your application. The X-Ray console uses the service graph to generate a visualization or *service map*.




For a distributed application, X-Ray combines nodes from all services that process requests with the same trace ID into a single service graph. The first service that the request hits adds a tracing header (p. 26) that is propagated between the front end and services that it calls.

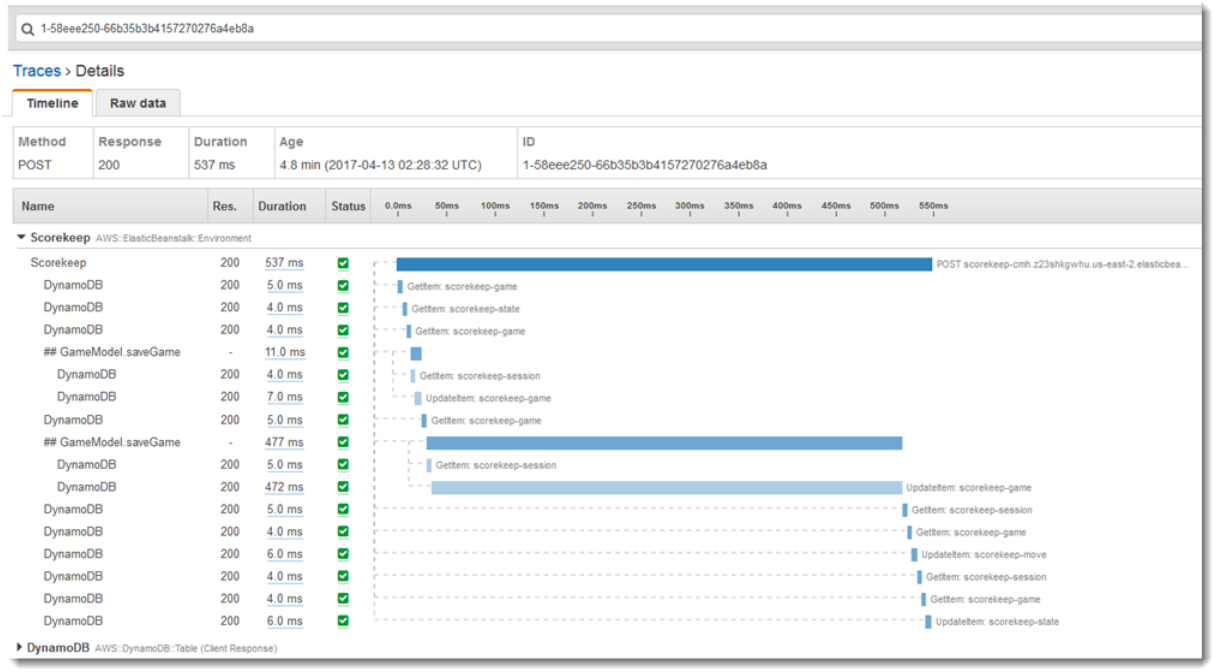
For example, Scorekeep (p. 115) runs a web API that calls a microservice (an AWS Lambda function) to generate a random name by using a Node.js library. The X-Ray SDK for Java generates the trace ID and includes it in calls to Lambda. Lambda sends tracing data and passes the trace ID to the function. The X-Ray SDK for Node.js also uses the trace ID to send data. As a result, nodes for the API, the Lambda service, and the Lambda function all appear as separate, but connected, nodes on the service map.




Service graph data is retained for 30 days.

# Traces

A **trace ID** tracks the path of a request through your application. A trace collects all the segments generated by a single request. That request is typically an HTTP GET or POST request that travels through a load balancer, hits your application code, and generates downstream calls to other AWS services or external web APIs. The first supported service that the HTTP request interacts with adds a trace ID header to the request, and propagates it downstream to track the latency, disposition, and other request data.




Service graph data is retained for 30 days.

# Sampling

To ensure efficient tracing and provide a representative sample of the requests that your application serves, the X-Ray SDK applies a **sampling** algorithm to determine which requests get traced. By default, the X-Ray SDK records the first request each second, and five percent of any additional requests.

To avoid incurring service charges when you are getting started, the default sampling rate is conservative. You can configure X-Ray to modify the default sampling rule and configure additional rules that apply sampling based on properties of the service or request.

For example, you might want to disable sampling and trace all requests for calls that modify state or handle user accounts or transactions. For high-volume read-only calls, like background polling, health checks, or connection maintenance, you can sample at a low rate and still get enough data to see any issues that arise.

For more information, see Configuring sampling rules in the X-Ray console (p. 70).

# Tracing header

All requests are traced, up to a configurable minimum. After reaching that minimum, a percentage of requests are traced to avoid unnecessary cost. The sampling decision and trace ID are added to HTTP requests in **tracing headers** named `X-Amzn-Trace-Id`. The first X-Ray-integrated service that the request hits adds a tracing header, which is read by the X-Ray SDK and included in the response.

**Example Tracing header with root trace ID and sampling decision**

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793;Sampled=1
```

**Tracing Header Security**
A tracing header can originate from the X-Ray SDK, an AWS service, or the client request. Your application can remove `X-Amzn-Trace-Id` from incoming requests to avoid issues caused by users adding trace IDs or sampling decisions to their requests.

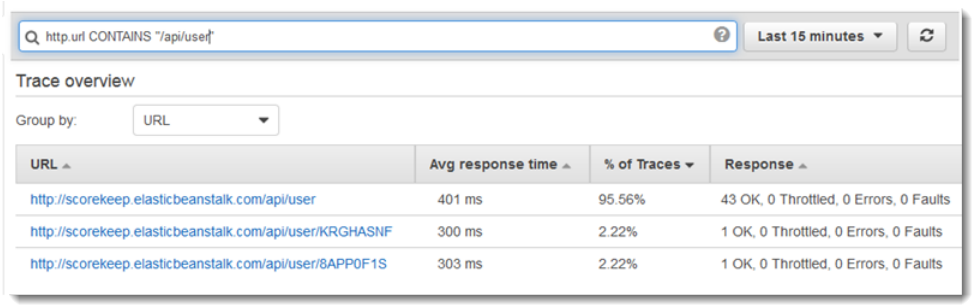
The tracing header can also contain a parent segment ID if the request originated from an instrumented application. For example, if your application calls a downstream HTTP web API with an instrumented HTTP client, the X-Ray SDK adds the segment ID for the original request to the tracing header of the downstream request. An instrumented application that serves the downstream request can record the parent segment ID to connect the two requests.

**Example Tracing header with root trace ID, parent segment ID and sampling decision**

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793;Parent=53995c3f42cd8ad8;Sampled=1
```

# Filter expressions

Even with sampling, a complex application generates a lot of data. The AWS X-Ray console provides an easy-to-navigate view of the service graph. It shows health and performance information that helps you identify issues and opportunities for optimization in your application. For advanced tracing, you can drill down to traces for individual requests, or use **filter expressions** to find traces related to specific paths or users.




# Groups

Extending filter expressions, X-Ray also supports the group feature. Using a filter expression, you can define criteria by which to accept traces into the group.

You can call the group by name or by Amazon Resource Name (ARN) to generate its own service graph, trace summaries, and Amazon CloudWatch metrics. Once a group is created, incoming traces are checked against the group's filter expression as they are stored in the X-Ray service. Metrics for the number of traces matching each criteria are published to CloudWatch every minute.

Updating a group's filter expression doesn't change data that's already recorded. The update applies only to subsequent traces. This can result in a merged graph of the new and old expressions. To avoid this, delete the current group and create a fresh one.

> **Note**
> Groups are billed by the number of retrieved traces that match the filter expression. For more information, see AWS X-Ray pricing.

# Annotations and metadata

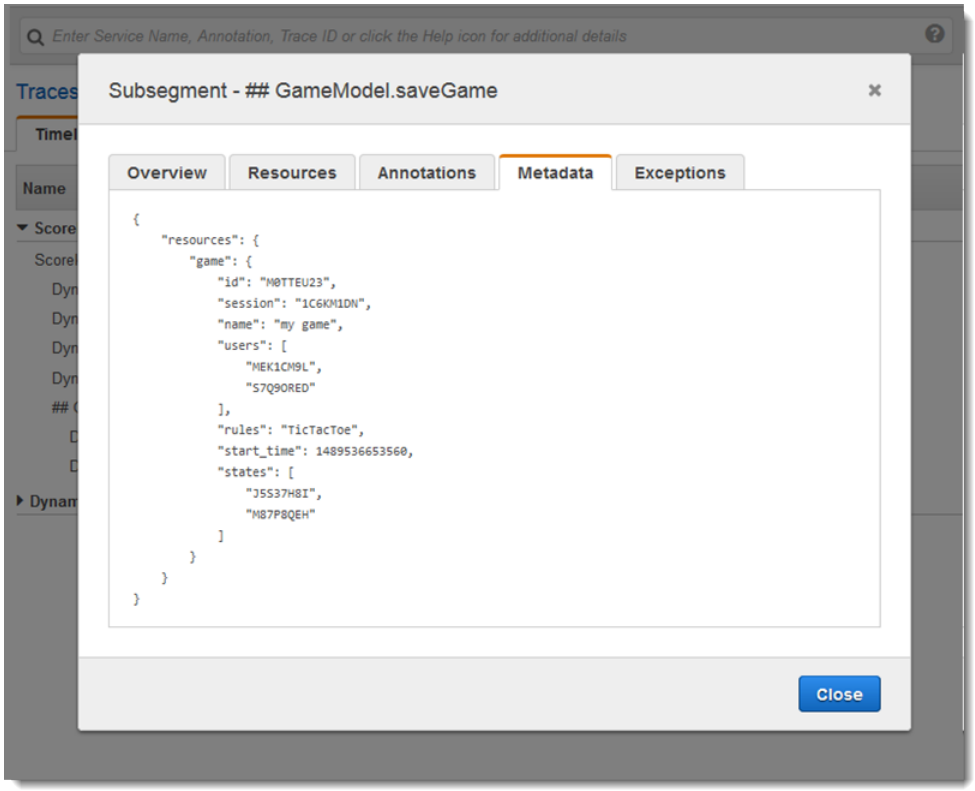
When you instrument your application, the X-Ray SDK records information about incoming and outgoing requests, the AWS resources used, and the application itself. You can add other information to the segment document as annotations and metadata. Annotations and metadata are aggregated at the trace level, and can be added to any segment or subsegment.

**Annotations** are simple key-value pairs that are indexed for use with filter expressions (p. 59). Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

X-Ray indexes up to 50 annotations per trace.

**Metadata** are key-value pairs with values of any type, including objects and lists, but that are not indexed. Use metadata to record data you want to store in the trace but don't need to use for searching traces.

You can view annotations and metadata in the segment or subsegment details in the X-Ray console.




# Errors, faults, and exceptions

X-Ray tracks errors that occur in your application code, and errors that are returned by downstream services. Errors are categorized as follows.

- **`Error`** – Client errors (400 series errors)
- **`Fault`** – Server faults (500 series errors)
- **`Throttle`** – Throttling errors (429 Too Many Requests)

When an exception occurs while your application is serving an instrumented request, the X-Ray SDK records details about the exception, including the stack trace, if available. You can view exceptions under segment details (p. 55) in the X-Ray console.

# Security in AWS X-Ray

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. The effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS compliance programs. To learn about the compliance programs that apply to X-Ray, 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 organization's requirements, and applicable laws and regulations.

This documentation will help you understand how to apply the shared responsibility model when using X-Ray. The following topics show you how to configure X-Ray to meet your security and compliance objectives. You'll also learn how to use other AWS services that can help you to monitor and secure your X-Ray resources.

**Topics**



# Data protection in AWS X-Ray

AWS X-Ray always encrypts traces and related data at rest. When you need to audit and disable encryption keys for compliance or internal requirements, you can configure X-Ray to use an AWS Key Management Service (AWS KMS) customer master key (CMK) to encrypt data.

X-Ray provides an AWS managed CMK named `aws/xray`. Use this key when you just want to audit key usage in AWS CloudTrail and don't need to manage the key itself. When you need to manage access to the key or configure key rotation, you can create a customer managed CMK.

When you change encryption settings, X-Ray spends some time generating and propagating data keys. While the new key is being processed, X-Ray may encrypt data with a combination of the new and old settings. Existing data is not re-encrypted when you change encryption settings.

> **Note**
> AWS KMS charges when X-Ray uses a CMK to encrypt or decrypt trace data.
>
> - **Default encryption** – Free.
> - **AWS managed CMK** – Pay for key use.
> - **Customer managed CMK** – Pay for key storage and use.

See AWS Key Management Service Pricing for details.

You must have user-level access to a customer managed CMK to configure X-Ray to use it, and to then view encrypted traces. See User permissions for encryption (p. 38) for more information.

**To configure X-Ray to use a CMK for encryption**

1. Open the X-Ray console.
2. Choose **Encryption**.
3. Choose **Use a customer master key**.
4. Choose a key from the dropdown menu:

   - **aws/xray** – Use the AWS managed CMK.
   - *key alias* – Use a customer managed CMK in your account.
   - **Manually enter a key ARN** – Use a customer managed CMK in a different account. Enter the full Amazon Resource Name (ARN) of the key in the field that appears.

   **Note**
   X-Ray does not support asymmetric CMKs.
5. Choose **Apply**.

If X-Ray is unable to access your encryption key, it stops storing data. This can happen if your user loses access to the CMK, or if you disable a key that's currently in use. When this happens, X-Ray shows a notification in the navigation bar.

To configure encryption settings with the X-Ray API, see Configuring sampling, groups, and encryption settings with the AWS X-Ray API (p. 95).

# Identity and access management for AWS X-Ray

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 X-Ray resources. IAM is an AWS service that you can use with no additional charge.

**Topics**



## Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work you do in X-Ray.

**Service user** – If you use the X-Ray service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more X-Ray 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 X-Ray, see Troubleshooting AWS X-Ray identity and access (p. 42).

**Service administrator** – If you're in charge of X-Ray resources at your company, you probably have full access to X-Ray. It's your job to determine which X-Ray features and resources your employees 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 X-Ray, see How AWS X-Ray works with IAM (p. 34).

**IAM administrator** – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to X-Ray. To view example X-Ray identity-based policies that you can use in IAM, see AWS X-Ray identity-based policy examples (p. 38).

# Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. For more information about signing in using the AWS Management Console, see The IAM Console and Sign-in Page in the *IAM User Guide*.

You must be *authenticated* (signed in to AWS) as the AWS account root user, an IAM user, or by assuming an IAM role. You can also use your company's single sign-on authentication, or even sign in using Google or Facebook. In these cases, your administrator previously set up identity federation using IAM roles. When you access AWS using credentials from another company, you are assuming a role indirectly.

To sign in directly to the AWS Management Console, use your password with your root user email or your IAM user name. You can access AWS programmatically using your root user or IAM user access keys. AWS provides SDK and command line tools to cryptographically sign your request using your credentials. If you don't use AWS tools, you must sign the request yourself. Do this using *Signature Version 4*, a protocol for authenticating inbound API requests. For more information about authenticating requests, see Signature Version 4 Signing Process in the *AWS General Reference*.

Regardless of the authentication method that you use, you might also 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 Using Multi-Factor Authentication (MFA) in AWS in the *IAM User Guide*.

## AWS Account Root User

When you first create an AWS account, you begin with a single 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 do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

## IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. An IAM user can have long-term credentials such as a user name and password or a set of access keys. To learn how to generate access keys, see Managing Access Keys for IAM Users in the *IAM User Guide*. When you generate access keys for an IAM user, make sure you view and securely save the key pair. You cannot recover the secret access key in the future. Instead, you must generate a new access key pair.

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 When to Create an IAM User (Instead of a Role) 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. You can temporarily assume an IAM role in the AWS Management Console by switching roles. 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 Using IAM Roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Temporary IAM user permissions** – An IAM user can assume an IAM role to temporarily take on different permissions for a specific task.
- **Federated user access** – Instead of creating an IAM user, you can use existing identities from AWS Directory Service, your enterprise user directory, or a web identity provider. These are known as *federated users*. AWS assigns a role to a federated user when access is requested through an identity provider. For more information about federated users, see Federated Users and Roles in the *IAM User Guide*.
- **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 How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.
- **AWS service access** – A service role is an IAM role that a service assumes to perform actions in your account on your behalf. When you set up some AWS service environments, you must define a role for the service to assume. This service role must include all the permissions that are required for the service to access the AWS resources that it needs. Service roles vary from service to service, but many allow you to choose your permissions as long as you meet the documented requirements for that service. Service roles provide access only within your account and cannot be used to grant access to services in other accounts. You can create, modify, and delete a service role from within IAM. For example, you can create a role that allows Amazon Redshift to access an Amazon S3 bucket on your behalf and then load data from that bucket into an Amazon Redshift cluster. For more information, see Creating a Role to Delegate Permissions to an AWS Service in the *IAM User Guide*.
- **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 Using an IAM Role to Grant Permissions to Applications Running on Amazon EC2 Instances in the *IAM User Guide*.

To learn whether to use IAM roles, see When to Create an IAM Role (Instead of a User) in the *IAM User Guide*.

# Managing access using policies

You control access in AWS by creating policies and attaching them to IAM identities or AWS resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions.

AWS evaluates these policies when an entity (root user, IAM user, or IAM role) 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*.

An IAM administrator can use policies to specify who has access to AWS resources, and what actions they can perform on those resources. Every IAM entity (user or role) starts with no permissions. In other words, by default, users can do nothing, not even change their own password. To give a user permission to do something, an administrator must attach a permissions policy to a user. Or the administrator can add the user to a group that has the intended permissions. When an administrator gives permissions to a group, all users in that group are granted those permissions.

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, role, or group. These policies control what actions that identity can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM 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 Choosing 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 such as an Amazon S3 bucket. Service administrators can use these policies to define what actions a specified principal (account member, user, or role) can perform on that resource and under what conditions. Resource-based policies are inline policies. There are no managed resource-based policies.

## Access control lists (ACLs)

Access control lists (ACLs) are a type of policy that controls 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 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 How SCPs Work 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 X-Ray works with IAM

Before you use IAM to manage access to X-Ray, you should understand what IAM features are available to use with X-Ray. To get a high-level view of how X-Ray and other AWS services work with IAM, see AWS Services That Work with IAM in the *IAM User Guide*.

You can use AWS Identity and Access Management (IAM) to grant X-Ray permissions to users and compute resources in your account. IAM controls access to the X-Ray service at the API level to enforce permissions uniformly, regardless of which client (console, AWS SDK, AWS CLI) your users employ.

To use the X-Ray console (p. 46) to view service maps and segments, you only need read permissions. To enable console access, add the `AWSXrayReadOnlyAccess` managed policy (p. 40) to your IAM user.

For local development and testing (p. 37), create an IAM user with read and write permissions. Generate access keys for the user and store them in the standard AWS SDK location. You can use these credentials with the X-Ray daemon, the AWS CLI, and the AWS SDK.

To deploy your instrumented app to AWS (p. 37), create an IAM role with write permissions and assign it to the resources running your application. `AWSXRayDaemonWriteAccess` includes permission to upload traces, and some read permissions as well to support the use of sampling rules (p. 70).

The read and write policies do not include permission to configure encryption key settings (p. 29) and sampling rules. Use `AWSXrayFullAccess` to access these settings, or add configuration APIs (p. 95) in a custom policy. For encryption and decryption with a customer managed key that you create, you also need permission to use the key (p. 38).

**Topics**

# X-Ray identity-based policies

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. X-Ray supports specific actions, resources, and condition keys. To learn about all of the elements that you use in a JSON policy, see IAM JSON Policy Elements Reference in the *IAM User Guide*.

## Actions

The `Action` element of an IAM identity-based policy describes the specific action or actions that will be allowed or denied by the policy. Policy actions usually have the same name as the associated AWS API operation. The action is used in a policy to grant permissions to perform the associated operation.

Policy actions in X-Ray use the following prefix before the action: `xray:`. For example, to grant someone permission to retrieve group resource details with the X-Ray `GetGroup` API operation, you include the `xray:GetGroup` action in their policy. Policy statements must include either an `Action` or `NotAction` element. X-Ray defines its own set of actions that describe tasks that you can perform with this service.

To specify multiple actions in a single statement, separate them with commas as follows:

```
"Action": [
      "xray:action1",
      "xray:action2"
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Get`, include the following action:

```
"Action": "xray:Get*"
```

To see a list of X-Ray actions, see Actions Defined by AWS X-Ray in the *IAM User Guide*.

## Resources

The `Resource` element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. You specify a resource using an ARN or using the wildcard (*) to indicate that the statement applies to all resources.

You can control access to resources by using an IAM policy. For actions that support resource-level permissions, you use an Amazon Resource Name (ARN) to identify the resource that the policy applies to.

All X-Ray actions can be used in an IAM policy to grant or deny users permission to use that action. However, not all X-Ray actions support resource-level permissions, which enable you to specify the resources on which an action can be performed.

For actions that don't support resource-level permissions, you must use "*" as the resource.

The following X-Ray actions support resource-level permissions:

- `CreateGroup`
- `GetGroup`
- `UpdateGroup`
- `DeleteGroup`
- `CreateSamplingRule`
- `UpdateSamplingRule`
- `DeleteSamplingRule`

The following is an example of an identity-based permissions policy for a `CreateGroup` action. The example shows the use of an ARN relating to Group name `local-users` with the unique ID as a wildcard. The unique ID is generated when the group is created, and so it can't be predicted in the policy in advance. When using `GetGroup`, `UpdateGroup`, or `DeleteGroup`, you can define this as either a wildcard or the exact ARN, including ID.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:CreateGroup"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:group/local-users/*"
            ]
        }
    ]
}
```

The following is an example of an identity-based permissions policy for a `CreateSamplingRule` action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:CreateSamplingRule"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:sampling-rule/base-scorekeep"
            ]
        }
    ]
}
```

**Note**
The ARN of a sampling rule is defined by its name. Unlike group ARNs, sampling rules have no uniquely generated ID.

To see a list of X-Ray resource types and their ARNs, see Resources Defined by AWS X-Ray in the *IAM User Guide*. To learn with which actions you can specify the ARN of each resource, see Actions Defined by AWS X-Ray.

### Condition keys

X-Ray does not provide any service-specific condition keys, but it does support using some global condition keys. To see all AWS global condition keys, see AWS Global Condition Context Keys in the *IAM User Guide*.

### Examples

To view examples of X-Ray identity-based policies, see AWS X-Ray identity-based policy examples (p. 38).

## X-Ray resource-based policies

X-Ray does not support resource-based policies.

## Authorization based on X-Ray tags

X-Ray does not support tagging resources or controlling access based on tags.

## Running your application locally

Your instrumented application sends trace data to the X-Ray daemon. The daemon buffers segment documents and uploads them to the X-Ray service in batches. The daemon needs write permissions to upload trace data and telemetry to the X-Ray service.

When you run the daemon locally (p. 142), store your IAM user's access key and secret key in a file named `credentials` in a folder named `.aws` in your user folder.

**Example ~/.aws/credentials**

```
[default]
aws_access_key_id=AKIAIOSFODNN7EXAMPLE
aws_secret_access_key=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

If you already configured credentials for use with the AWS SDK or AWS CLI, the daemon can use those. If multiple profiles are available, the daemon uses the default profile.

## Running your application in AWS

When you run your application on AWS, use a role to grant permission to the Amazon EC2 instance or Lambda function that runs the daemon.

- **Amazon Elastic Compute Cloud (Amazon EC2)** – Create an IAM role and attach it to the EC2 instance as an instance profile.
- **Amazon Elastic Container Service (Amazon ECS)** – Create an IAM role and attach it to container instances as a container instance IAM role.
- **AWS Elastic Beanstalk (Elastic Beanstalk)** – Elastic Beanstalk includes X-Ray permissions in its default instance profile. You can use the default instance profile, or add write permissions to a custom instance profile.
- **AWS Lambda (Lambda)** – Add write permissions to your function's execution role.

**To create a role for use with X-Ray**

1. Open the IAM console.
2. Choose **Roles**.
3. Choose **Create New Role**.
4. For **Role Name**, type **`xray-application`**. Choose **Next Step.**
5. For **Role Type**, choose **Amazon EC2**.
6. Attach managed policies to give your application access to AWS services.

   - **AWSXRayDaemonWriteAccess** – Gives the X-Ray daemon permission to upload trace data.
   - **AmazonS3ReadOnlyAccess** (Amazon EC2 only) – Gives the instance permission to download the X-Ray daemon from Amazon S3.

   If your application uses the AWS SDK to access other services, add policies that grant access to those services.
7. Choose **Next Step**.
8. Choose **Create Role**.

## User permissions for encryption

X-Ray encrypts all trace data and by default, and you can configure it to use a key that you manage (p. 29). If you choose a AWS Key Management Service customer managed customer master key (CMK), you need to ensure that the key's access policy lets you grant permission to X-Ray to use it to encrypt. Other users in your account also need access to the key to view encrypted trace data in the X-Ray console.

For a customer managed CMK, configure your key with an access policy that allows the following actions:

- User who configures the key in X-Ray has permission to call `kms:CreateGrant` and `kms:DescribeKey`.
- Users who can access encrypted trace data have permission to call `kms:Decrypt`.

When you add a user to the **Key users** group in the key configuration section of the IAM console, they have permission for both of these operations. Permission only needs to be set on the key policy, so you don't need any AWS KMS permissions on your IAM users, groups, or roles. For more information, see Using Key Policies in the AWS KMS Developer Guide.

For default encryption, or if you choose the AWS managed CMK (`aws/xray`), permission is based on who has access to X-Ray APIs. Anyone with access to `PutEncryptionConfig`, included in `AWSXrayFullAccess`, can change the encryption configuration. To prevent a user from changing the encryption key, do not give them permission to use `PutEncryptionConfig`.

# AWS X-Ray identity-based policy examples

By default, IAM users and roles don't have permission to create or modify X-Ray resources. They also can't perform tasks using the AWS Management Console, AWS CLI, or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform specific API operations on the specified resources they need. The administrator must then attach those policies to the IAM users or groups that require those permissions.

To learn how to create an IAM identity-based policy using these example JSON policy documents, see Creating Policies on the JSON Tab in the *IAM User Guide*.

**Topics**



## Policy best practices

Identity-based policies are very powerful. They determine whether someone can create, access, or delete X-Ray 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 Using AWS Managed Policies** – To start using X-Ray quickly, use AWS managed policies to give your employees the permissions they need. These policies are already available in your account and are maintained and updated by AWS. For more information, see Get Started Using Permissions With AWS Managed Policies in the *IAM User Guide*.
- **Grant Least Privilege** – When you create custom policies, grant only the permissions required to perform a task. Start with a minimum set of permissions and grant additional permissions as

necessary. Doing so is more secure than starting with permissions that are too lenient and then trying to tighten them later. For more information, see Grant Least Privilege in the *IAM User Guide*.

- **Enable MFA for Sensitive Operations** – For extra security, require IAM users to use multi-factor authentication (MFA) to access sensitive resources or API operations. For more information, see Using Multi-Factor Authentication (MFA) in AWS in the *IAM User Guide*.
- **Use Policy Conditions for Extra Security** – To the extent that it's practical, define the conditions under which your identity-based policies allow access to a resource. For example, you can write conditions to specify a range of allowable IP addresses that a request must come from. You can also write conditions to allow requests only within a specified date or time range, or to require the use of SSL or MFA. For more information, see IAM JSON Policy Elements: Condition in the *IAM User Guide*.

## Using the X-Ray console

To access the AWS X-Ray console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the X-Ray 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 (IAM users or roles) with that policy.

To ensure that those entities can still use the X-Ray console, also attach the following AWS managed policy to the entities. For more information, see Adding Permissions to a User in the *IAM User Guide*:

```
AWSXrayReadOnlyAccess
```

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 you're trying to perform.

## Allow 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": "*"
        }
    ]
}
```

## IAM managed policies for X-Ray

To make granting permissions easy, IAM supports **managed policies** for each service. A service can update these managed policies with new permissions when it releases new APIs. AWS X-Ray provides managed policies (p. 40) for read only, write only, and administrator use cases.

- `AWSXrayReadOnlyAccess` – Read permissions for using the X-Ray console, AWS CLI, or AWS SDK to get trace data and service maps from the X-Ray API. Includes permission to view sampling rules.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:GetSamplingRules",
                "xray:GetSamplingTargets",
                "xray:GetSamplingStatisticSummaries",
                "xray:BatchGetTraces",
                "xray:GetServiceGraph",
                "xray:GetTraceGraph",
                "xray:GetTraceSummaries",
                "xray:GetGroups",
                "xray:GetGroup"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

- `AWSXRayDaemonWriteAccess` – Write permissions for using the X-Ray daemon, AWS CLI, or AWS SDK to upload segment documents and telemetry to the X-Ray API. Includes read permissions to get sampling rules (p. 70) and report sampling results.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingRules",
                "xray:GetSamplingTargets",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

- `AWSXrayFullAccess` – Permission to use all X-Ray APIs, including read permissions, write permissions, and permission to configure encryption key settings and sampling rules.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:*"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

**To add a managed policy to an IAM user, group, or role**

1. Open the IAM console.
2. Open the role associated with your instance profile, an IAM user, or an IAM group.
3. Under **Permissions**, attach the managed policy.

## Specifying a resource within an IAM policy

You can control access to resources by using an IAM policy. For actions that support resource-level permissions, you use an Amazon Resource Name (ARN) to identify the resource that the policy applies to.

All X-Ray actions can be used in an IAM policy to grant or deny users permission to use that action. However, not all X-Ray actions support resource-level permissions, which enable you to specify the resources on which an action can be performed.

For actions that don't support resource-level permissions, you must use "*" as the resource.

The following X-Ray actions support resource-level permissions:

- `CreateGroup`
- `GetGroup`
- `UpdateGroup`
- `DeleteGroup`
- `CreateSamplingRule`
- `UpdateSamplingRule`
- `DeleteSamplingRule`

The following is an example of an identity-based permissions policy for a `CreateGroup` action. The example shows the use of an ARN relating to Group name `local-users` with the unique ID as a wildcard. The unique ID is generated when the group is created, and so it can't be predicted in the policy in advance. When using `GetGroup`, `UpdateGroup`, or `DeleteGroup`, you can define this as either a wildcard or the exact ARN, including ID.

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

```
            "Effect": "Allow",
            "Action": [
                "xray:CreateGroup"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:group/local-users/*"
            ]
        }
    ]
}
```

The following is an example of an identity-based permissions policy for a `CreateSamplingRule` action.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:CreateSamplingRule"
            ],
            "Resource": [
                "arn:aws:xray:eu-west-1:123456789012:sampling-rule/base-scorekeep"
            ]
        }
    ]
}
```

**Note**
The ARN of a sampling rule is defined by its name. Unlike group ARNs, sampling rules have no uniquely generated ID.

# Troubleshooting AWS X-Ray identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with X-Ray and IAM.

**Topics**



## I Am not authorized to perform an action in X-Ray

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 sampling rule but does not have `xray:GetSamplingRules` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 xray:GetSamplingRules on resource: arn:${Partition}:xray:${Region}:${Account}:sampling-
rule/${SamplingRuleName}
```

In this case, Mateo asks his administrator to update his policies to allow him to access the sampling rule resource using the `xray:GetSamplingRules` 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, then you must contact your administrator for assistance. Your administrator is the person that provided you with your user name and password. Ask that person to update your policies to allow you to pass a role to X-Ray.

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 X-Ray. However, the action requires the service to have permissions 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 asks her administrator to update her policies to allow her to perform the `iam:PassRole` action.

## I want to view my access keys

After you create your IAM user access keys, you can view your access key ID at any time. However, you can't view your secret access key again. If you lose your secret key, you must create a new access key pair.

Access keys consist of two parts: an access key ID (for example, `AKIAIOSFODNN7EXAMPLE`) and a secret access key (for example, `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`). Like a user name and password, you must use both the access key ID and secret access key together to authenticate your requests. Manage your access keys as securely as you do your user name and password.

> **Important**
> Do not provide your access keys to a third party, even to help find your canonical user ID. By doing this, you might give someone permanent access to your account.

When you create an access key pair, you are prompted to save the access key ID and secret access key in a secure location. The secret access key is available only at the time you create it. If you lose your secret access key, you must add new access keys to your IAM user. You can have a maximum of two access keys. If you already have two, you must delete one key pair before creating a new one. To view instructions, see Managing Access Keys in the *IAM User Guide*.

## I'm an administrator and want to allow others to access X-Ray

To allow others to access X-Ray, you must create an IAM entity (user or role) for the person or application that needs access. They will use the credentials for that entity to access AWS. You must then attach a policy to the entity that grants them the correct permissions in X-Ray.

To get started right away, see Creating Your First IAM Delegated User and Group in the *IAM User Guide*.

## I want to allow people outside of my AWS account to access my X-Ray 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 X-Ray supports these features, see How AWS X-Ray works with IAM (p. 34).
- 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 How IAM Roles Differ from Resource-based Policies in the *IAM User Guide*.

# Logging and monitoring in AWS X-Ray

Monitoring is an important part of maintaining the reliability, availability, and performance of your AWS solutions. You should collect monitoring data from all of the parts of your AWS solution so that you can more easily debug a multi-point failure if one occurs. AWS provides several tools for monitoring your X-Ray resources and responding to potential incidents:

**AWS CloudTrail Logs**

AWS X-Ray integrates with AWS CloudTrail to record API actions made by a user, a role, or an AWS service in X-Ray. You can use CloudTrail to monitor X-Ray API requests in real time and store logs in Amazon S3, Amazon CloudWatch Logs, and Amazon CloudWatch Events. For more information, see Logging X-Ray API calls with AWS CloudTrail (p. 156).

**AWS Config Tracking**

AWS X-Ray integrates with AWS Config to record configuration changes made to your X-Ray encryption resources. You can use AWS Config to inventory X-Ray encryption resources, audit the X-Ray configuration history, and send notifications based on resource changes. For more information, see Tracking X-Ray encryption configuration changes with AWS Config (p. 163).

**Amazon CloudWatch Monitoring**

You can use the X-Ray SDK for Java to publish unsampled Amazon CloudWatch metrics from your collected X-Ray segments. These metrics are derived from the segment's start and end time, and the error, fault and throttled status flags. Use these trace metrics to expose retries and dependency issues within subsegments. For more information, see AWS X-Ray metrics for the X-Ray SDK for Java (p. 208).

# Compliance validation for AWS X-Ray

Third-party auditors assess the security and compliance of AWS X-Ray as part of multiple AWS compliance programs. These include SOC, PCI, HIPAA, and others.

For a list of AWS services in scope of specific compliance programs, see AWS Services in Scope by Compliance Program. 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 X-Ray 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 and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying security- and compliance-focused baseline environments on AWS.
- Architecting for HIPAA Security and Compliance Whitepaper – This whitepaper describes how companies can use AWS to create HIPAA-compliant applications.
- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Config – This AWS 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 that helps you check your compliance with security industry standards and best practices.

# Resilience in AWS X-Ray

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 Availability Zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

# Infrastructure security in AWS X-Ray

As a managed service, AWS X-Ray 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 X-Ray 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.

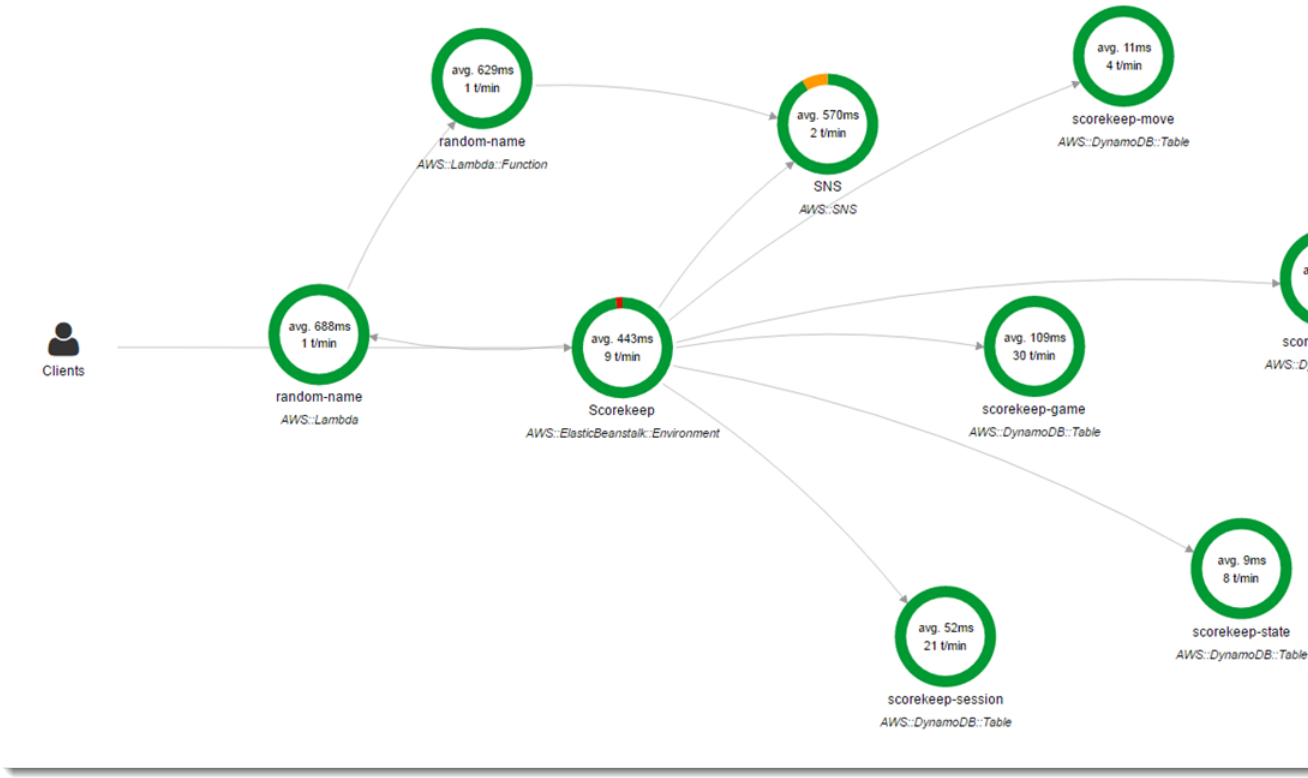
Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the AWS Security Token Service (AWS STS) to generate temporary security credentials to sign requests.

# AWS X-Ray console

The AWS X-Ray console enables you to view service maps and traces for requests that your applications serve.

The console's service map is a visual representation of the JSON service graph that X-Ray generates from the trace data generated by your applications.

The map consists of service nodes for each application in your account that serves requests, upstream client nodes that represent the origins of the requests, and downstream service nodes that represent web services and resources used by an application while processing a request.



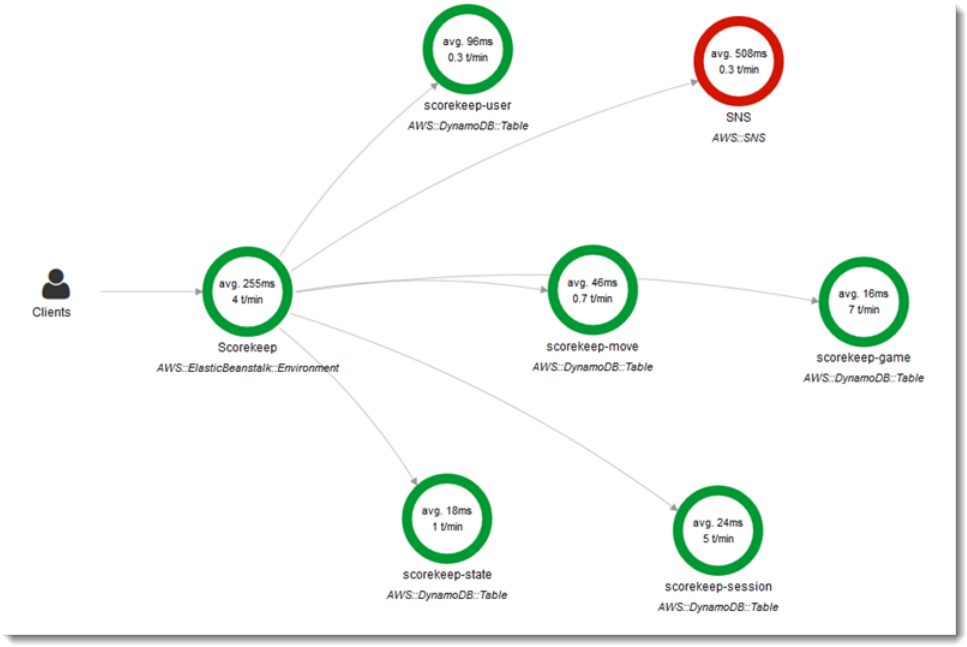

You can use filters to view a service map or traces for a specific request, service, connection between two services (an edge), or requests that satisfy a condition. X-Ray provides a filter expression language for filtering requests, services, and edges based on data in request headers, response status, and indexed fields on the original segments.

## Viewing the service map

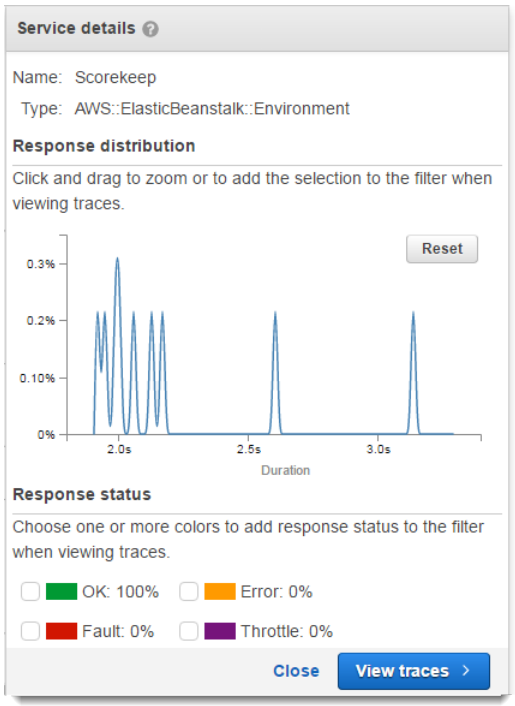
View the service map in the X-Ray console to identify services where errors are occurring, connections with high latency, or traces for requests that were unsuccessful.

**To view the service map**

1. Open the service map page of the X-Ray console.



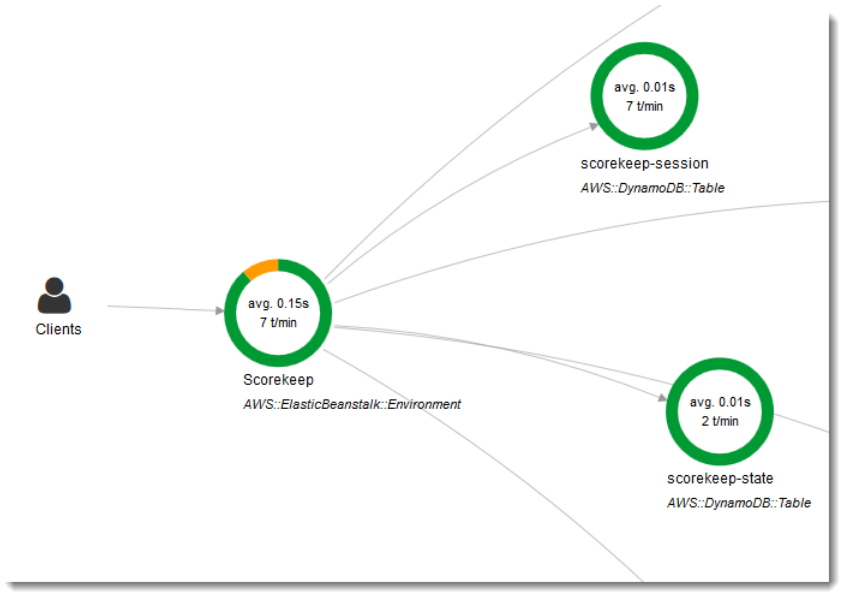

2. Choose a service node to view requests for that node, or an edge between two nodes to view requests that traveled that connection.
3. Use the histogram (p. 68) to filter traces by duration, and select status codes for which you want to view traces. Then choose **View traces** to open the trace list with the filter expression applied.




The service map indicates the health of each node by coloring it based on the ratio of successful calls to errors and faults:

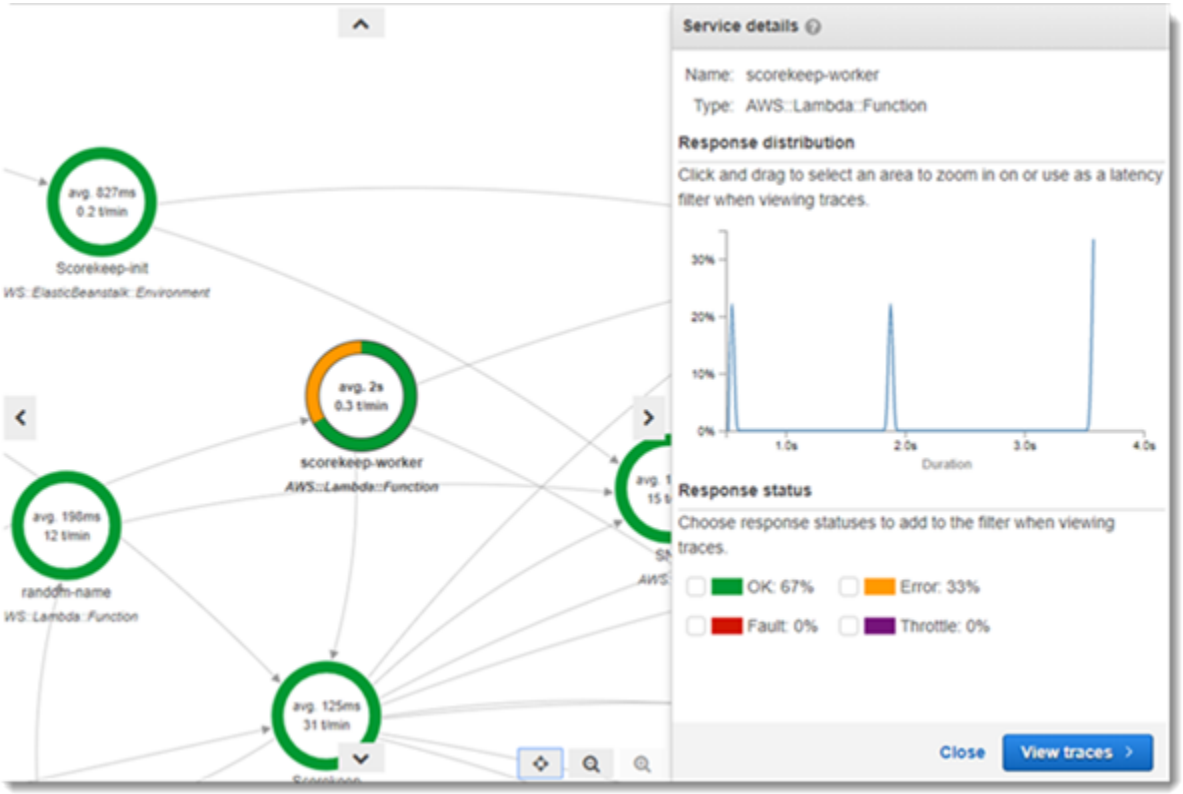
- **Green** for successful calls

- **Red** for server faults (500 series errors)
- **Yellow** for client errors (400 series errors)
- **Purple** for throttling errors (429 Too Many Requests)




In the center of each node, the console shows the average response time and number of traces that it sent per minute during the chosen time range.

If your service map is large, the console defaults to a zoomed out view. Use the on-screen controls or mouse to zoom in and out and move the image around.




**Controls**

-  – Zoom in or out. You can also use the mouse wheel to zoom in or out.
-  – Scroll the service map. Click and drag to scroll by using the mouse.
-  – Frame the selected node or edge in the center of map.

# Viewing the service map by group

Using a filter expression, you can define criteria by which to accept traces into a group. Use the following steps to then display that specific group in the service map.

**To view a group service map**

1. Open the service map page of the X-Ray console.
2. Choose a group name from the dropdown menu to the left of the search bar.


# Changing the node presentation

Use the following options to change the way your service map is presented. You can toggle on service icons to better follow the stream of services, and you can change the weights of your nodes to better represent traffic or health.

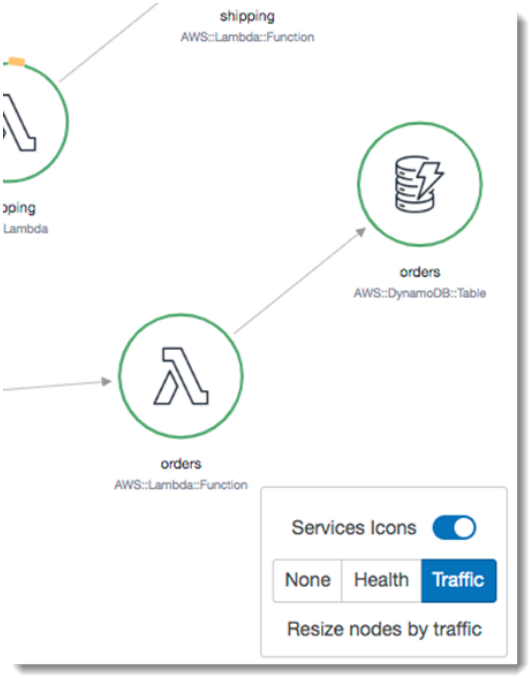
**To change the node presentation**

1. Open the service map page of the X-Ray console, or an individual trace map.

   Node options are in the bottom right of the map.
2. Choose a presentation for your node.

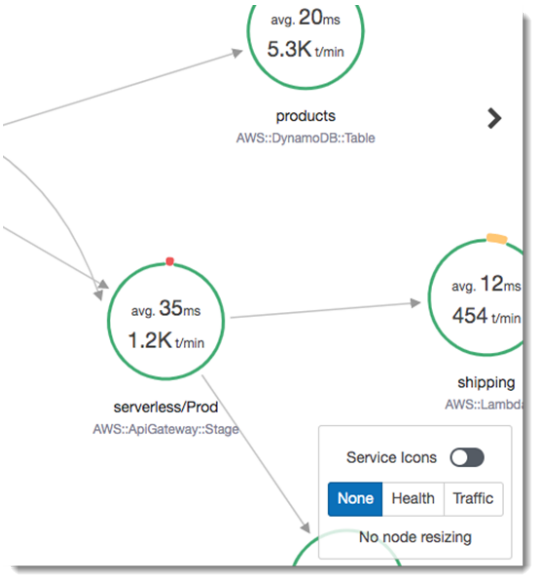
   Service Icons enabled

   **Service Icons** – When enabled, shows an icon for the AWS service respresented by the node, instead of the default summary of activity.



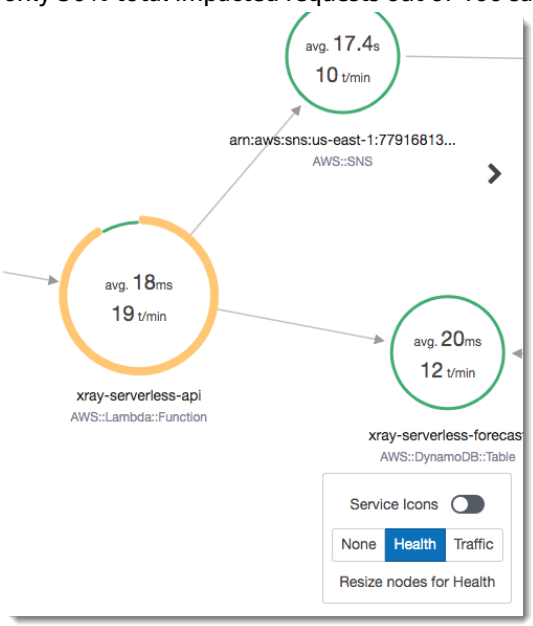
Node weight by None

**None** – No node resizing, all nodes are of the same weight.




Node weight by Health

**Health** – The node size is calculated by the total number of impacted requests. Impacted requests include `fault`, `error`, and `throttle`. For example, a node with 10% total impacted requests out of 1000 sampled (100 requests impacted) has a larger node size than a node with only 50% total impacted requests out of 100 sampled (50 requests impacted).




Node weight by Traffic

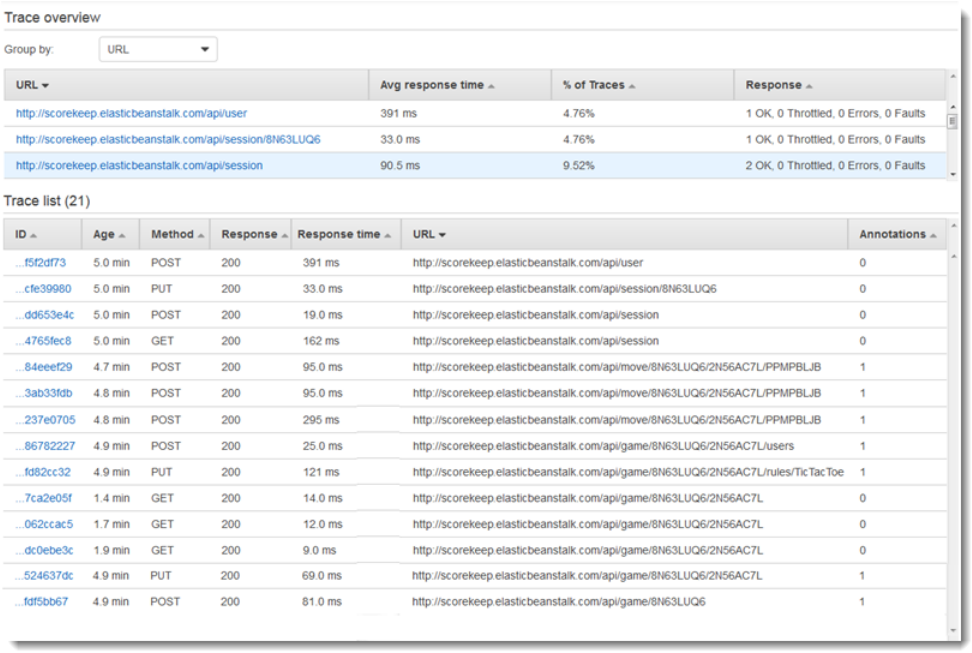
**Traffic** – The node size is calculated by the total number of sampled requests. For example, a node with 1000 sampled requests has a larger node size than one with 100 sampled requests.


# Viewing traces

Use the trace list in the X-Ray console to find traces by URL, response code, or other data from the trace summary.

### To use the trace list

1. Open the Trace overview in the X-Ray console.



Trace overview

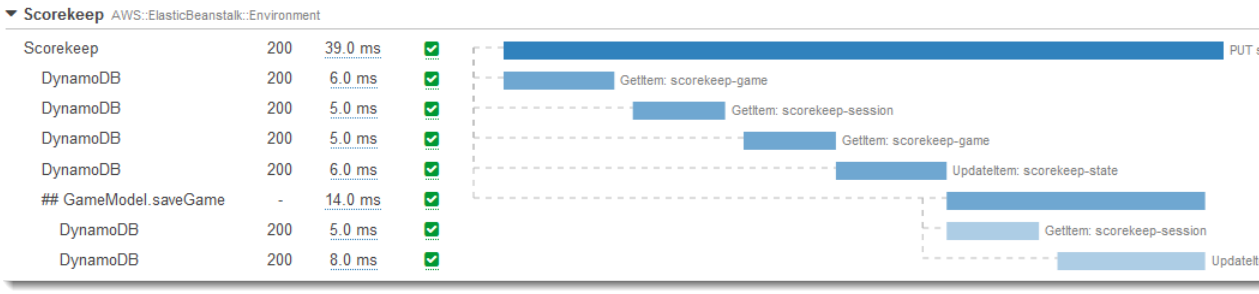
Group by: URL

| URL | Avg response time | % of Traces | Response |
|---|---|---|---|
| http://scorekeep.elasticbeanstalk.com/api/user | 391 ms | 4.76% | 1 OK, 0 Throttled, 0 Errors, 0 Faults |
| http://scorekeep.elasticbeanstalk.com/api/session/8N63LUQ6 | 33.0 ms | 4.76% | 1 OK, 0 Throttled, 0 Errors, 0 Faults |
| http://scorekeep.elasticbeanstalk.com/api/session | 90.5 ms | 9.52% | 2 OK, 0 Throttled, 0 Errors, 0 Faults |

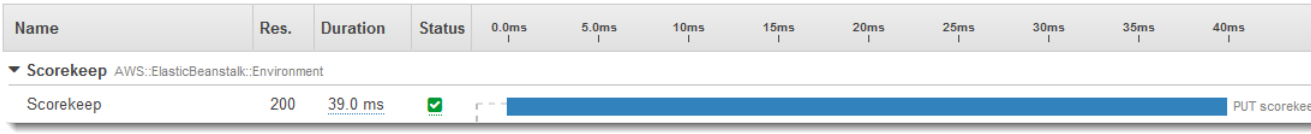
Trace list (21)

| ID | Age | Method | Response | Response time | URL | Annotations |
|---|---|---|---|---|---|---|
| ...f5f2df73 | 5.0 min | POST | 200 | 391 ms | http://scorekeep.elasticbeanstalk.com/api/user | 0 |
| ...cfe39980 | 5.0 min | PUT | 200 | 33.0 ms | http://scorekeep.elasticbeanstalk.com/api/session/8N63LUQ6 | 0 |
| ...dd653e4c | 5.0 min | POST | 200 | 19.0 ms | http://scorekeep.elasticbeanstalk.com/api/session | 0 |
| ...4765fec8 | 5.0 min | GET | 200 | 162 ms | http://scorekeep.elasticbeanstalk.com/api/session | 0 |
| ...84eeef29 | 4.7 min | POST | 200 | 95.0 ms | http://scorekeep.elasticbeanstalk.com/api/move/8N63LUQ6/2N56AC7L/PPMPBLJB | 1 |
| ...3ab33fdb | 4.8 min | POST | 200 | 95.0 ms | http://scorekeep.elasticbeanstalk.com/api/move/8N63LUQ6/2N56AC7L/PPMPBLJB | 1 |
| ...237e0705 | 4.8 min | POST | 200 | 295 ms | http://scorekeep.elasticbeanstalk.com/api/move/8N63LUQ6/2N56AC7L/PPMPBLJB | 1 |
| ...86782227 | 4.9 min | POST | 200 | 25.0 ms | http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L/users | 1 |
| ...fd82cc32 | 4.9 min | PUT | 200 | 121 ms | http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L/rules/TicTacToe | 1 |
| ...7ca2e05f | 1.4 min | GET | 200 | 14.0 ms | http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L | 0 |
| ...062ccac5 | 1.7 min | GET | 200 | 12.0 ms | http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L | 0 |
| ...dc0ebe3c | 1.9 min | GET | 200 | 9.0 ms | http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L | 0 |
| ...524637dc | 4.9 min | PUT | 200 | 69.0 ms | http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6/2N56AC7L | 1 |
| ...fdf5bb67 | 4.9 min | POST | 200 | 81.0 ms | http://scorekeep.elasticbeanstalk.com/api/game/8N63LUQ6 | 1 |

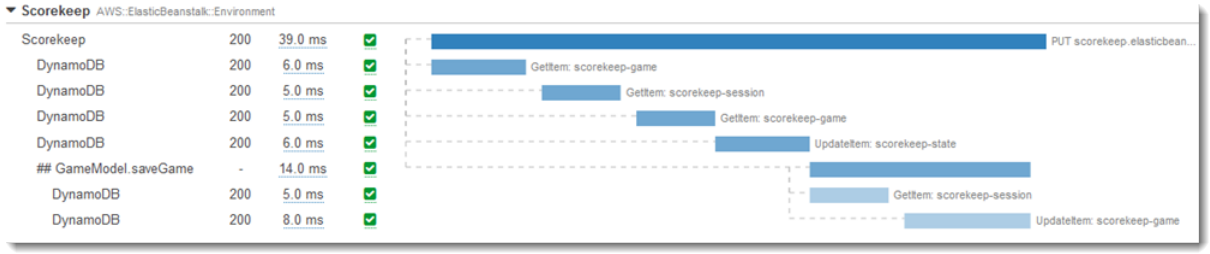
2. Choose a URL to filter the trace list.
3. Choose a trace ID to view the timeline for a trace.




The **Timeline** view shows a hierarchy of segments and subsegments. The first entry in the list is the segment, which represents all data recorded by the service for a single request.

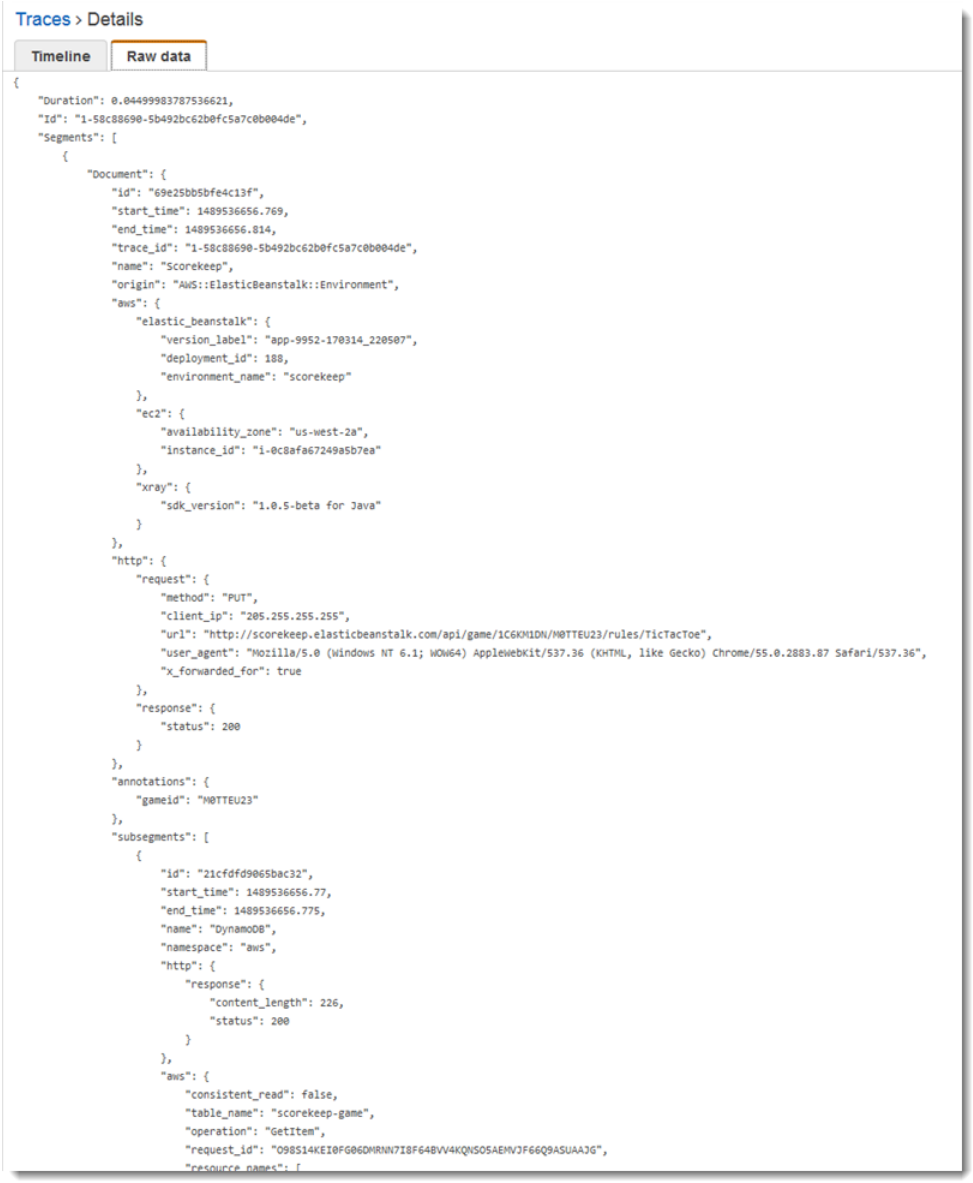



Below the segment are subsegments. This example shows subsegments recorded by instrumented Amazon DynamoDB clients, and a custom subsegment.




The X-Ray SDK records subsegments automatically when you use an instrumented AWS SDK, HTTP, or SQL client to make calls to external resources. You can also tell the SDK to record custom subsegments for any function or block of code. Additional subsegments recorded while a custom subsegment is open become children of the custom subsegment.

From the **Timeline** view, you can also access the raw trace data that the console uses to generate the timeline. Choose **Raw data** to see the JSON document that contains all the segments and subsegments that make up the trace.

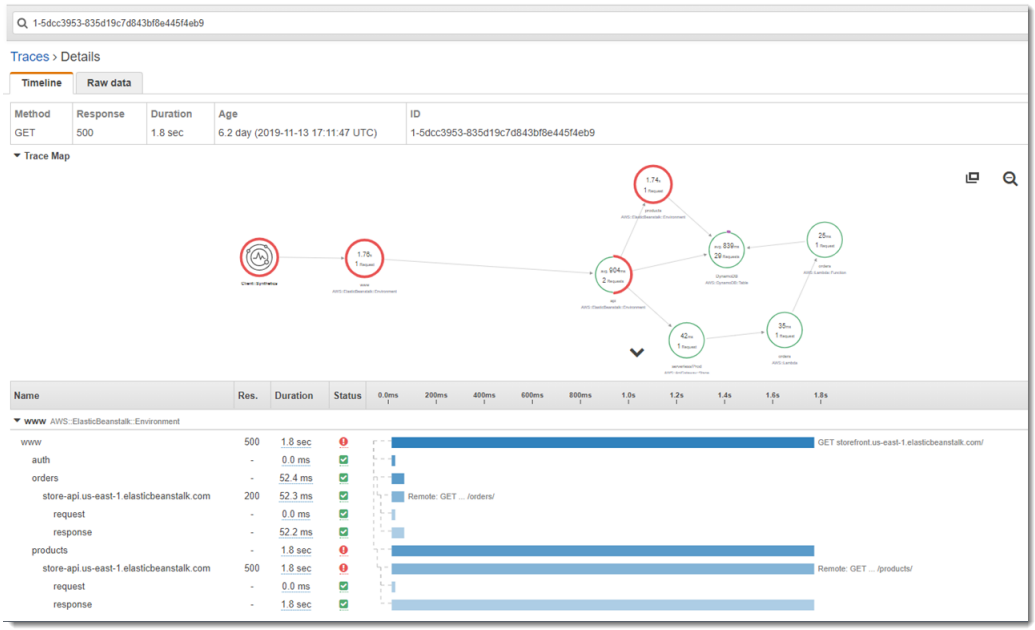



# Viewing the trace map

Use the trace map to visually map the end-to-end path of a single request.

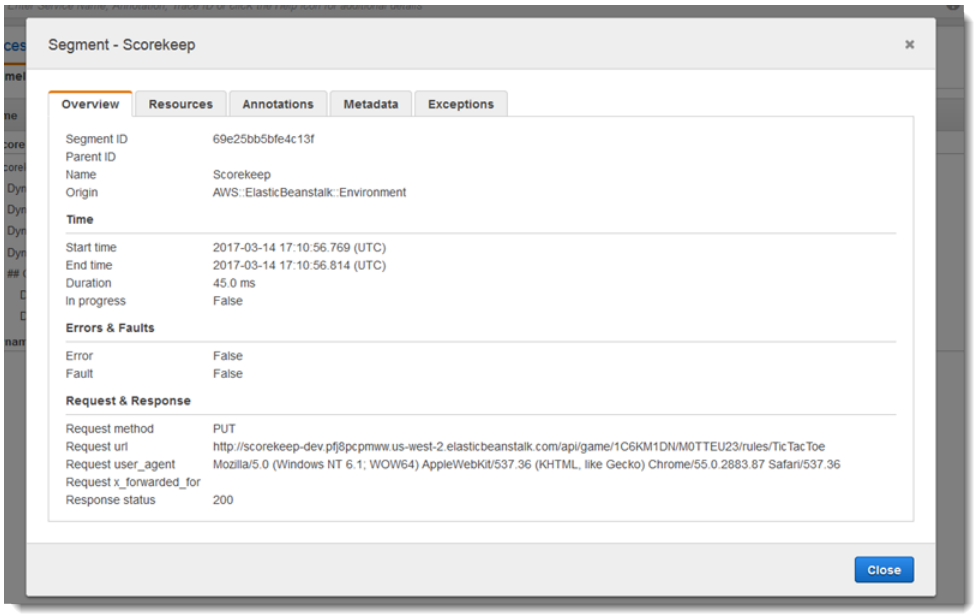
### To view a trace map

1. Open the Trace overview in the X-Ray console.
2. Choose a URL to filter the trace list.
3. Choose a trace ID to view the timeline for a trace. The trace map is displayed above the trace timeline.

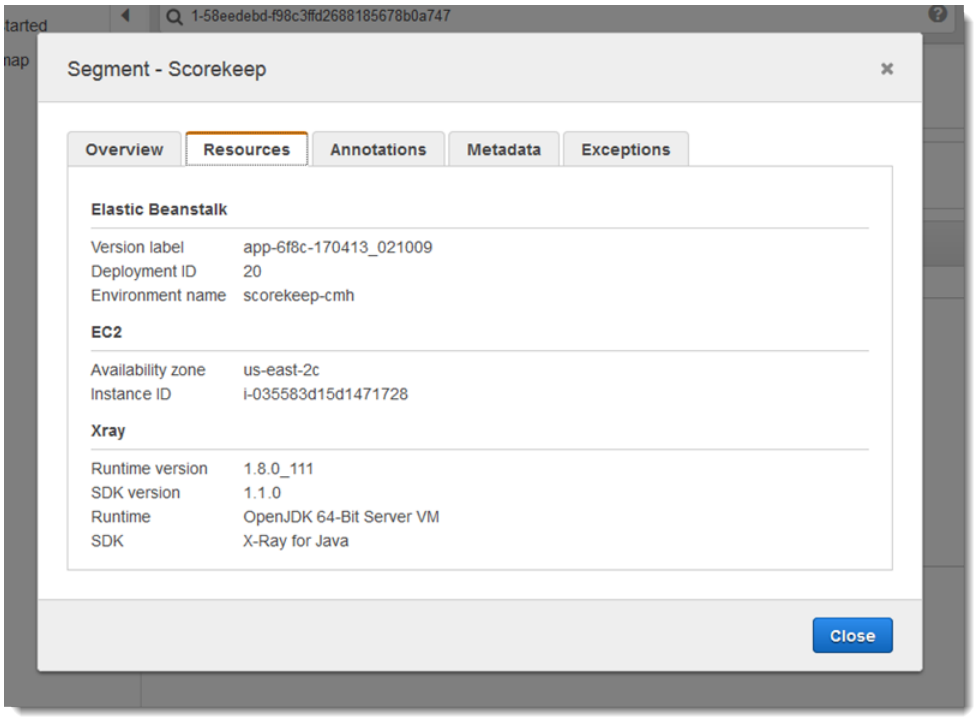



# Viewing segment details

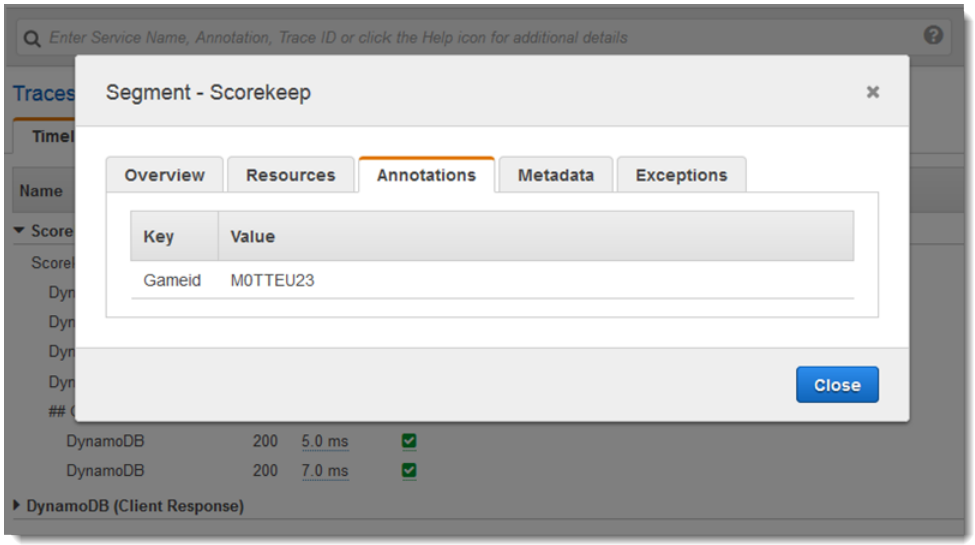
From the trace timeline, choose the name of a segment to view its details. The **Overview** tab shows information about the request and response.




The **Resources** tab for a segment shows information about the AWS resources running your application and the X-Ray SDK. Use the Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS plugin for the SDK to record service-specific resource information.

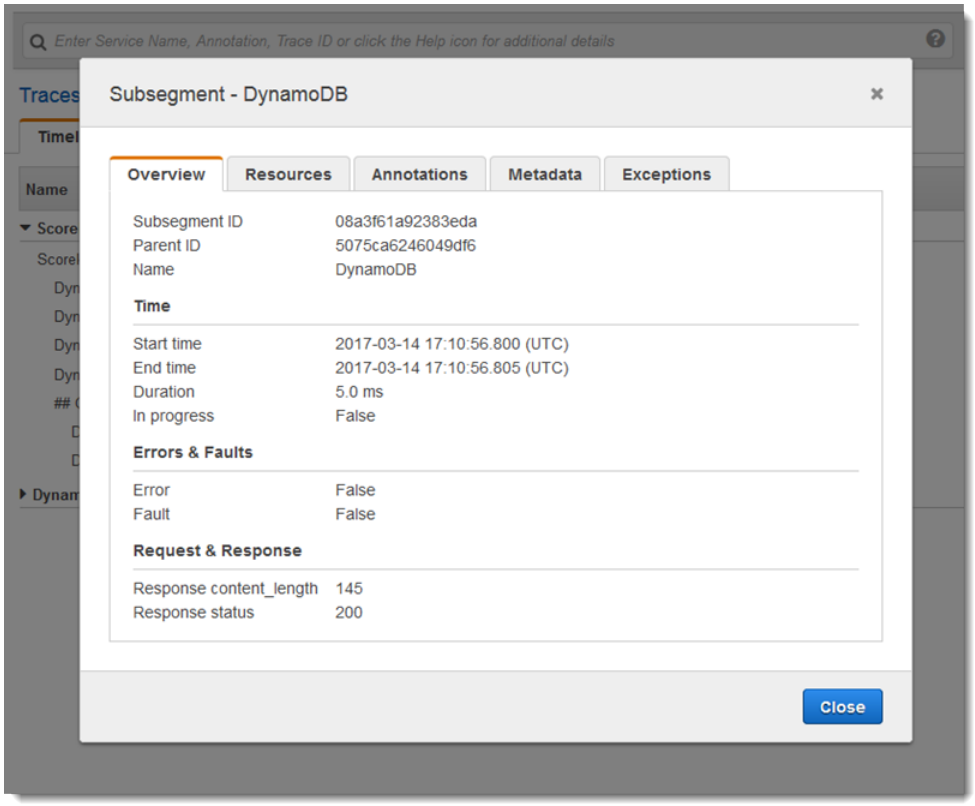



The remaining tabs show **Annotations**, **Metadata**, and **Exceptions** recorded on the segment. Exceptions are captured automatically when thrown from an instrumented request. Annotations and metadata contain additional information that you record by using the methods that the SDK provides.

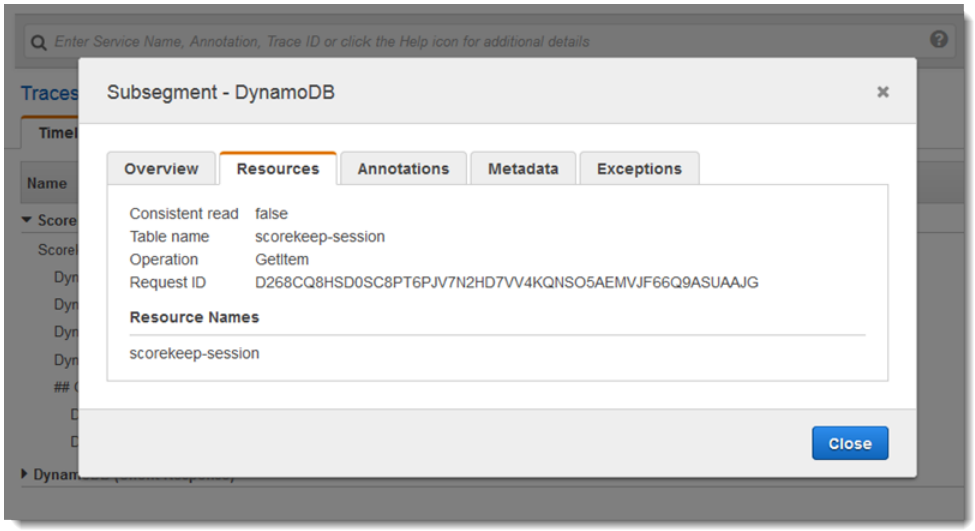



# Viewing subsegment details

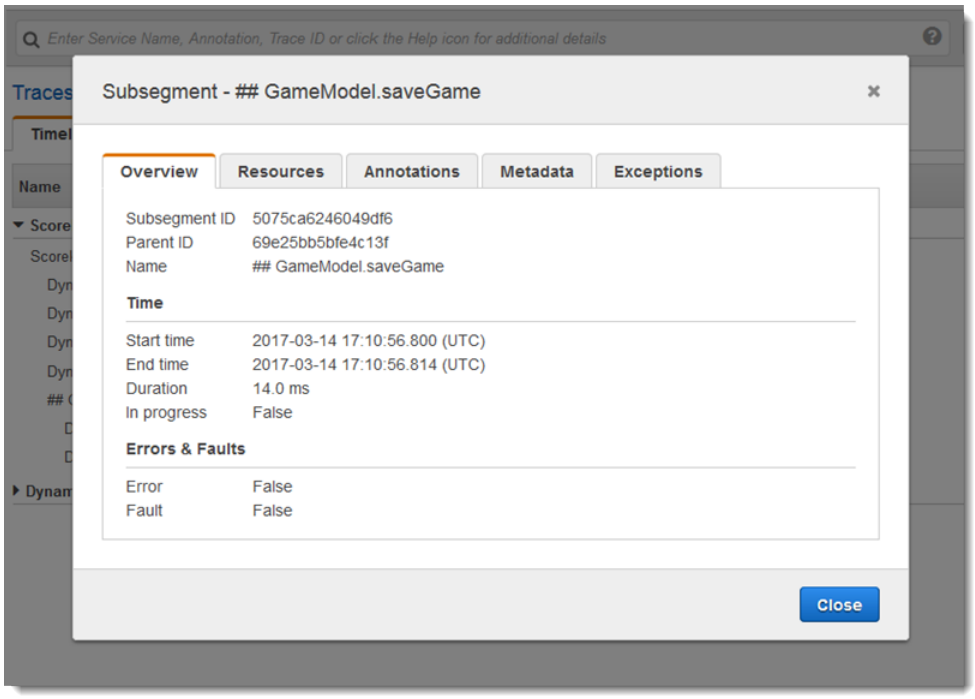
From the trace timeline, choose the name of a segment to view its details. For subsegments generated with instrumented clients, the **Overview** tab contains information about the request and response from your application's point of view. This example shows a subsegment from an instrumented call to DynamoDB.



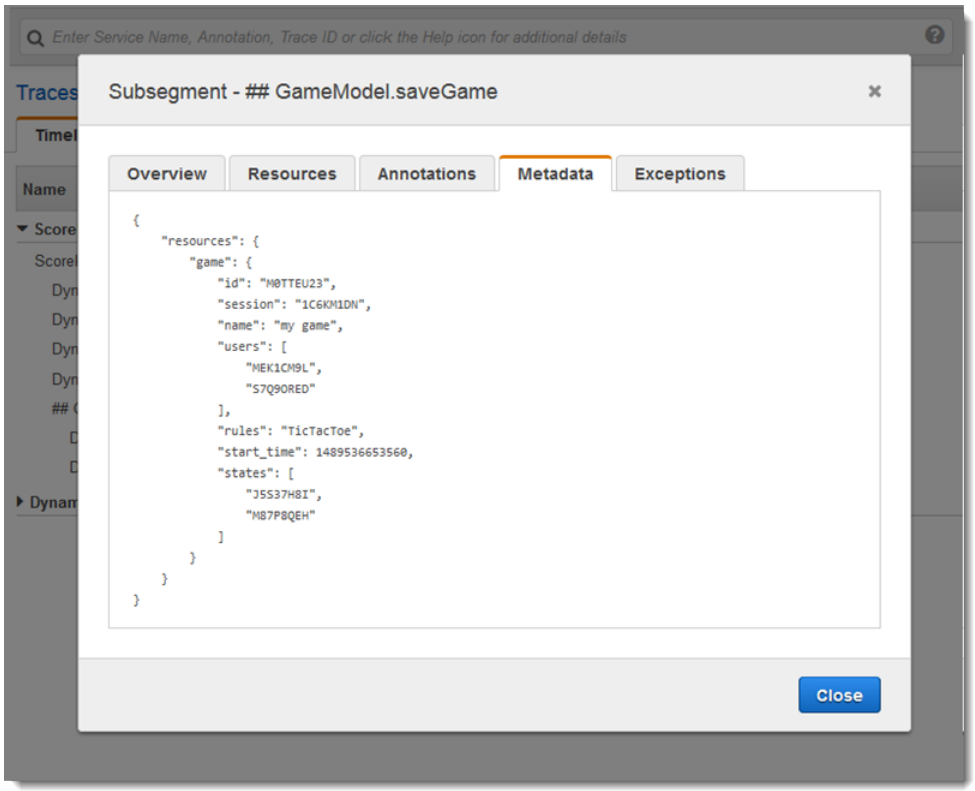

The **Resources** tab for a subsegment shows details about the DynamoDB table, operation called, and request ID.




For custom subsegments, the **Overview** tab shows the name of the subsegment, which you can set to specify the area of the code or function that it records.




Use custom subsegments to organize subsegments from instrumented clients into groups. You can also record metadata and annotations on subsegments, which can help you debug functions.



In this example, the application records the state of each `Game` object that it saves to DynamoDB. It does this by passing the object into the `putMetadata` method on the subsegment. The X-Ray SDK serializes the object into JSON and adds it to the segment document.

# Using filter expressions to search for traces in the console

When you choose a time period of traces to view in the X-Ray console, you might get more results than the console can display. In the upper-right corner, the console shows the number of traces that it scanned and whether there are more traces available.

You can narrow the results to just the traces that you want to find by using a *filter expression*.

**Topics**



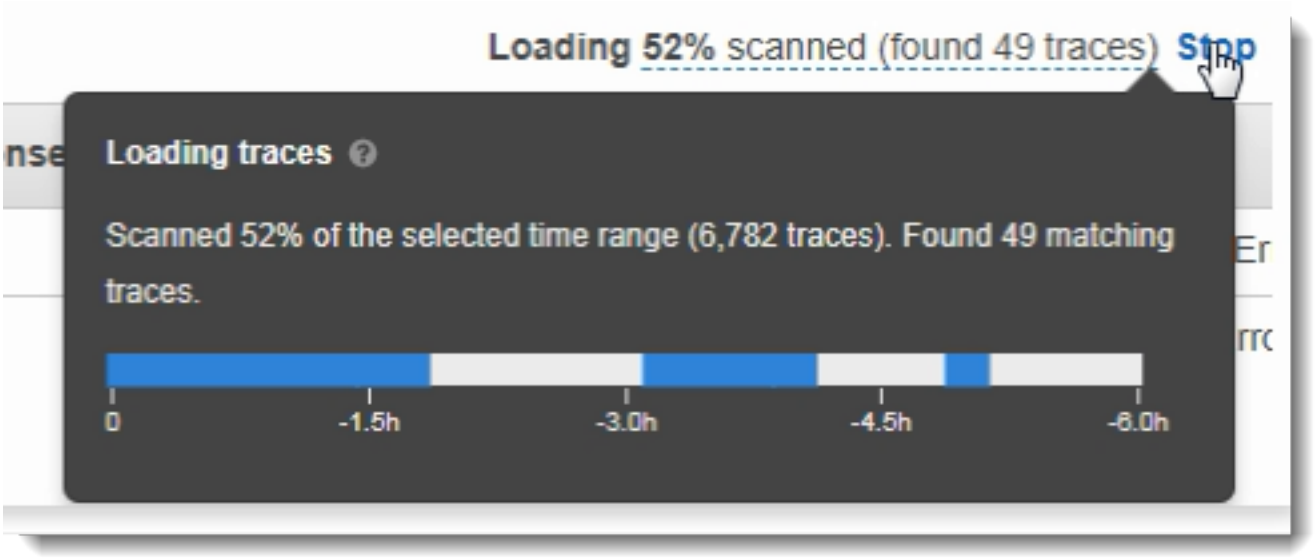
## Filter expression details

When you choose a node in the service map (p. 46), the console constructs a filter expression based on the service name of the node, and the types of error present based on your selection. To find traces that show performance issues or that relate to specific requests, you can adjust the expression that the console provides or create your own. If you add annotations with the X-Ray SDK, you can also filter based on the presence of an annotation key or the value of a key.

> **Note**
> If you choose a relative time range in the service map and choose a node, the console converts the time range to an absolute start and end time. To ensure that the traces for the node appear in the search results, and avoid scanning times when the node wasn't active, the time range only includes times when the node sent traces. To search relative to the current time, you can switch back to a relative time range in the traces page and scan again.

If there are still more results available than the console can show, the console shows you how many traces matched and the number of traces scanned. The percentage shown is the percentage of the selected time frame that was scanned. To ensure that you see all matching traces represented in the results, narrow your filter expression further, or choose a shorter time frame.

To get the freshest results first, the console starts scanning at the end of the time range and works backward. If there are a large number of traces, but few results, the console splits the time range into chunks and scans them in parallel. The progress bar shows the parts of the time range that have been scanned.




# Using filter expressions with groups

Groups are a collection of traces that are defined by a filter expression. You can use groups to generate additional service graphs and supply Amazon CloudWatch metrics.

Groups are identified by their name or an Amazon Resource Name (ARN), and contain a filter expression. The service compares incoming traces to the expression and stores them accordingly.

You can create and modify groups by using the dropdown menu to the left of the filter expression search bar.

> **Note**
> If the service encounters an error in qualifying a group, that group is no longer included in processing incoming traces and an error metric is recorded.

# Filter expression syntax

Filter expressions can contain a *keyword*, a unary or binary *operator*, and a *value* for comparison.

```
keyword operator value
```

Different operators are available for different types of keyword. For example, `responsetime` is a number keyword and can be compared with operators related to numbers.

**Example – requests where response time was greater than 5 seconds**

```
responsetime > 5
```

You can combine multiple expressions in a compound expression by using the `AND` or `OR` operators.

**Example – requests where the total duration was 5–8 seconds**

```
duration >= 5 AND duration <= 8
```

Simple keywords and operators find issues only at the trace level. If an error occurs downstream, but is handled by your application and not returned to the user, a search for `error` will not find it.

To find traces with downstream issues, you can use the complex keywords (p. 64) `service()` and `edge()`. These keywords let you apply a filter expression to all downstream nodes, a single downstream node, or an edge between two nodes. For more granularity, you can filter services and edges by type with the id() function (p. 65).

# Boolean keywords

Boolean keyword values are either true or false. Use these keywords to find traces that resulted in errors.

### Boolean keywords

- `ok` – Response status code was 2XX Success.
- `error` – Response status code was 4XX Client Error.
- `throttle` – Response status code was 429 Too Many Requests.
- `fault` – Response status code was 5XX Server Error.
- `partial` – Request has incomplete segments.
- `inferred` – Request has inferred segments.
- `first` – Element is the first of an enumerated list.
- `last` – Element is the last of an enumerated list.
- `remote` – Root cause entity is remote.
- `root` – Service is the entry point or root segment of a trace.

Boolean operators find segments where the specified key is `true` or `false`.

### Boolean operators

- none – The expression is true if the keyword is true.
- `!` – The expression is true if the keyword is false.
- `=`,`!=` – Compare the value of the keyword to the string `true` or `false`. These operators act the same as the other operators but are more explicit.

### Example – response status is 2XX OK

```
ok
```

### Example – response status is not 2XX OK

```
!ok
```

### Example – response status is not 2XX OK

```
ok = false
```

### Example – last enumerated fault trace has error name "deserialize"

```
rootcause.fault.entity { last and name = "deserialize" }
```

**Example – requests with remote segments where coverage is greater than 0.7 and the service name is "traces"**

```
rootcause.responsetime.entity { remote and coverage > 0.7 and name = "traces" }
```

**Example – requests with inferred segments where the service type is "AWS:DynamoDB"**

```
rootcause.fault.service { inferred and name = traces and type = "AWS::DynamoDB" }
```

**Example – requests that have a segment with the name "data-plane" as the root**

```
service("data-plane") {root = true and fault = true}
```

# Number keywords

Use number keywords to search for requests with a specific response time, duration, or response status.

### Number keywords

- `responsetime` – Time that the server took to send a response.
- `duration` – Total request duration, including all downstream calls.
- `http.status` – Response status code.
- `index` – Position of an element in an enumerated list.
- `coverage` – Decimal percentage of entity response time over root segment response time. Applicable only for response time root cause entities.

### Number operators

Number keywords use standard equality and comparison operators.

- =, ! = – The keyword is equal to or not equal to a number value.
- <,<=, >,>= – The keyword is less than or greater than a number value.

**Example – response status is not 200 OK**

```
http.status != 200
```

**Example – request where the total duration was 5–8 seconds**

```
duration >= 5 AND duration <= 8
```

**Example – requests that completed successfully in less than 3 seconds, including all downstream calls**

```
ok !partial duration <3
```

**Example – enumerated list entity that has an index greater than 5**

```
rootcause.fault.service { index > 5 }
```

**Example – requests where the last entity that has coverage greater than 0.8**

```
rootcause.responsetime.entity { last and coverage > 0.8 }
```

# String keywords

Use string keywords to find traces with specific text in the request headers, or specific user IDs.

**String keywords**

- `http.url` – Request URL.
- `http.method` – Request method.
- `http.useragent` – Request user agent string.
- `http.clientip` – Requestor's IP address.
- `user` – Value of the user field on any segment in the trace.
- `name` – The name of a service or exception.
- `type` – Service type.
- `message` – Exception message.
- `availabilityzone` – Value of the availabilityzone field on any segment in the trace.
- `instance.id` – Value of the instance ID field on any segment in the trace.
- `resource.arn` – Value of the resource ARN field on any segment in the trace.

String operators find values that are equal to or contain specific text. Values must always be specified in quotation marks.

**String operators**

- `=`, `!=` – The keyword is equal to or not equal to a number value.
- `CONTAINS` – The keyword contains a specific string.
- `BEGINSWITH` , `ENDSWITH` – The keyword begins or ends with a specific string.

**Example – http.url filter**

```
http.url CONTAINS "/api/game/"
```

To test if a field exists on a trace, regardless of its value, check to see if it contains the empty string.

**Example – user filter**

Find all traces with user IDs.

```
user CONTAINS ""
```

**Example – select traces with a fault root cause that includes a service named "Auth"**

```
rootcause.fault.service { name = "Auth" }
```

**Example – select traces with a response time root cause whose last service has a type of DynamoDB**

```
rootcause.responsetime.service { last and type = "AWS::DynamoDB" }
```

**Example – select traces with a fault root cause whose last exception has the message "access denied for account_id: 1234567890"**

```
rootcause.fault.exception { last and message = "Access Denied for account_id: 1234567890"
```

# Complex keywords

Use complex keywords to find requests based on service name, edge name, or annotation value. For services and edges, you can specify an additional filter expression that applies to the service or edge. For annotations, you can filter on the value of an annotation with a specific key using Boolean, number, or string operators.

### Complex keywords

- `service(`*`name`*`) {`*`filter`*`}` – Service with name *`name`*. Optional curly braces can contain a filter expression that applies to segments created by the service.
- `edge(`*`source`*`, `*`destination`*`) {`*`filter`*`}` – Connection between services *`source`* and *`destination`*. Optional curly braces can contain a filter expression that applies to segments on this connection.
- `annotation.`*`key`* – Value of an annotation with field *`key`*. The value of an annotation can be a Boolean, number, or string, so you can use any of the comparison operators of those types. You can't use this keyword in combination with the `service` or `edge` keyword.
- `json` – JSON root cause object. See Getting data from AWS X-Ray (p. 86) for steps to create JSON entities programmatically.

Use the service keyword to find traces for requests that hit a certain node on your service map.

### Example – Service filter

Requests that included a call to `api.example.com` with a fault (500 series error).

```
service("api.example.com") { fault }
```

You can exclude the service name to apply a filter expression to all nodes on your service map.

### Example – service filter

Requests that caused a fault anywhere on your service map.

```
service() { fault }
```

The edge keyword applies a filter expression to a connection between two nodes.

### Example – edge filter

Request where the service `api.example.com` made a call to `backend.example.com` that failed with an error.

```
edge("api.example.com", "backend.example.com") { error }
```

You can also use the `!` operator with service and edge keywords to exclude a service or edge from the results of another filter expression.

**Example – service and request filter**

Request where the URL begins with `http://api.example.com/` and contains `/v2/` but does not reach a service named `api.example.com`.

```
http.url BEGINSWITH "http://api.example.com/" AND http.url CONTAINS "/v2/" AND !
service("api.example.com")
```

For annotations, use the comparison operators that correspond to the type of value.

**Example – annotation with string value**

Requests with an annotation named `gameid` with string value `"817DL6VO"`.

```
annotation.gameid = "817DL6VO"
```

**Example – annotation with number value**

Requests with annotation age with numerical value greater than 29.

```
annotation.age > 29
```

**Example – JSON with root cause entity**

Requests with matching root cause entities.

```
rootcause.json = #[{ "Services": [ { "Name": "GetWeatherData", "EntityPath": [{ "Name":
 "GetWeatherData" }, { "Name": "get_temperature" } ] }, { "Name": "GetTemperature",
 "EntityPath": [ { "Name": "GetTemperature" } ] } ] }]
```

# id function

When you provide a service name to the `service` or `edge` keyword, you get results for all nodes that have that name. For more precise filtering, you can use the `id` function to specify a service type in addition to a name to distinguish between nodes with the same name.

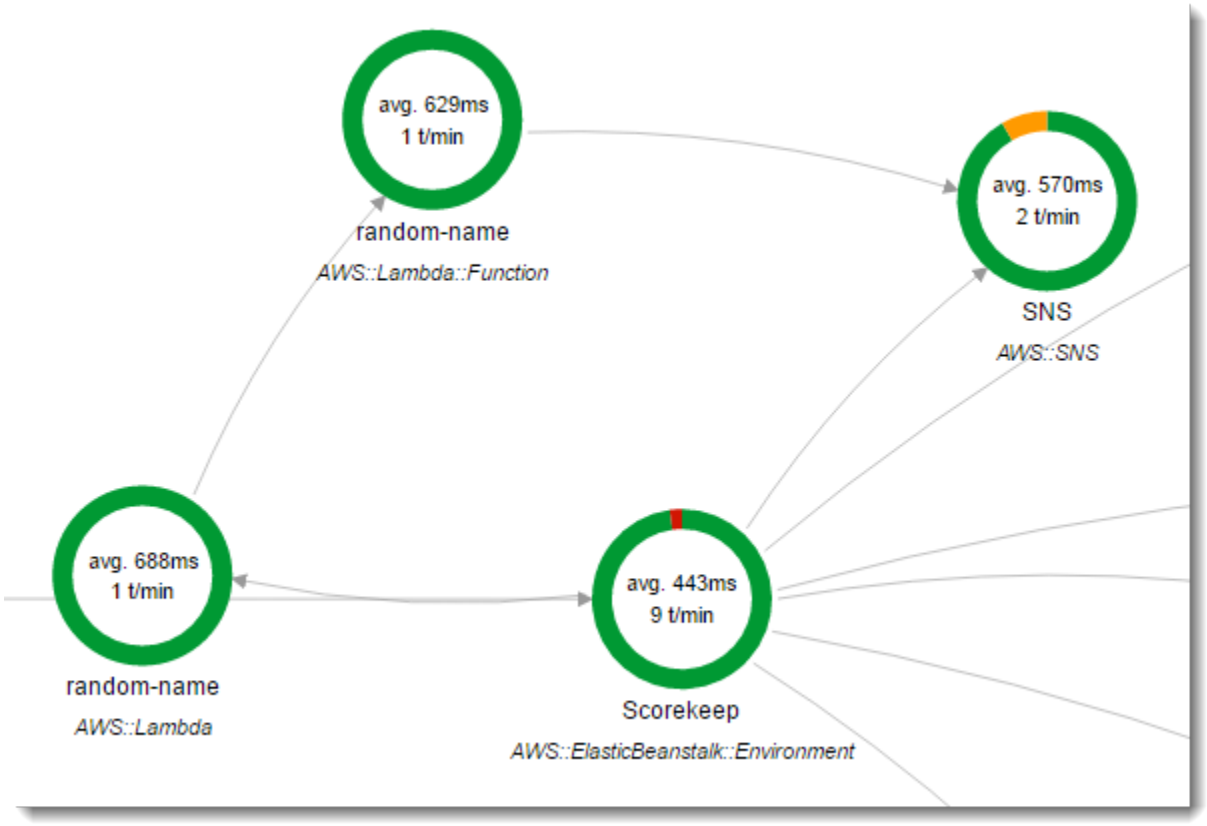
```
id(name: "service-name", type:"service::type")
```

You can use the `id` function in place of a service name in service and edge filters.

```
service(id(name: "service-name", type:"service::type")) { filter }
```

```
edge(id(name: "service-one", type:"service::type"), id(name: "service-two",
 type:"service::type")) { filter }
```

For example, the Scorekeep sample application (p. 115) includes an AWS Lambda function named `random-name`. This creates two nodes in the service map, one for the function invocation, and one for the Lambda service.




The two nodes have the same name but different types. A standard service filter will find traces for both.

### Example – service filter

Requests that include an error on any service named `random-name`.

```
service("random-name") { error }
```

Use the `id` function to narrow the search to errors on the function itself, excluding errors from the service.

### Example – service filter with id function

Requests that include an error on a service named `random-name` with type `AWS::Lambda::Function`.

```
service(id(name: "random-name", type: "AWS::Lambda::Function")) { error }
```

To search for nodes by type, you can also exclude the name entirely.

### Example – service filter with id function

Requests that include an error on a service with type `AWS::Lambda::Function`.

```
service(id(type: "AWS::Lambda::Function")) { error }
```

# Deep linking

You can use routes and queries to deep link into specific traces, or filtered views of traces and the service map.

**Console pages**

- Welcome page – xray/home#/welcome
- Getting started – xray/home#/getting-started
- Service map – xray/home#/service-map
- Traces – xray/home#/traces

## Traces

You can generate links for timeline, raw, and map views of individual traces.

**Trace timeline** – `xray/home#/traces/`*`trace-id`*

**Raw trace data** – `xray/home#/traces/`*`trace-id`*`/raw`

**Example – raw trace data**

```
https://console.aws.amazon.com/xray/home#/traces/1-57f5498f-d91047849216d0f2ea3b6442/raw
```

## Filter expressions

Link to a filtered list of traces.

**Filtered traces view** – `xray/home#/traces?filter=`*`filter-expression`*

**Example – filter expression**

```
https://console.aws.amazon.com/xray/home#/traces?filter=service("api.amazon.com") { fault =
 true OR responsetime > 2.5 } AND annotation.foo = "bar"
```

**Example – filter expression (URL encoded)**

```
https://console.aws.amazon.com/xray/home#/traces?filter=service(%22api.amazon.com%22)%20%7B
%20fault%20%3D%20true%20OR%20responsetime%20%3E%202.5%20%7D%20AND%20annotation.foo%20%3D
%20%22bar%22
```

For more information about filter expressions, see Using filter expressions to search for traces in the console (p. 59).

## Time range

Specify a length of time or start and end time in ISO8601 format. Time ranges are in UTC and can be up to 6 hours long.

**Length of time** – `xray/home#/`*`page`*`?timeRange=`*`range-in-minutes`*

**Example – service map for the last hour**

```
https://console.aws.amazon.com/xray/home#/service-map?timeRange=PT1H
```

**Start and end time** – `xray/home#/page?timeRange=start~end`

**Example – time range accurate to seconds**

```
https://console.aws.amazon.com/xray/home#/traces?
timeRange=2018-9-01T16:00:00~2018-9-01T22:00:00
```

**Example – time range accurate to minutes**

```
https://console.aws.amazon.com/xray/home#/traces?timeRange=2018-9-01T16:00~2018-9-01T22:00
```

## Region

Specify an AWS Region to link to pages in that Region. If you don't specify a Region, the console redirects you to the last visited Region.

**Region** – `xray/home?region=region#/page`

**Example – service map in US West (Oregon) (us-west-2)**

```
https://console.aws.amazon.com/xray/home?region=us-west-2#/service-map
```

When you include a Region with other query parameters, the Region query goes before the hash, and the X-Ray-specific queries go after the page name.

**Example – service map for the last hour in US West (Oregon) (us-west-2)**

```
https://console.aws.amazon.com/xray/home?region=us-west-2#/service-map?timeRange=PT1H
```

## Combined

**Example – recent traces with a duration filter**

```
https://console.aws.amazon.com/xray/home#/traces?timeRange=PT15M&filter=duration%20%3E%3D
%205%20AND%20duration%20%3C%3D%208
```

**Output**

- Page – Traces
- Time Range – Last 15 minutes
- Filter – duration >= 5 AND duration <= 8

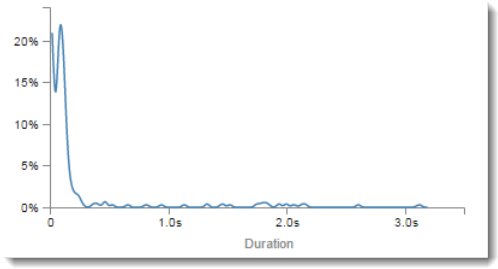
# Using latency histograms in the X-Ray console

When you select a node or edge on an AWS X-Ray service map (p. 46), the X-Ray console shows a latency distribution histogram.

# Latency

Latency is the amount of time between when a request starts and when it completes. A histogram shows a distribution of latencies. It shows duration on the x-axis, and the percentage of requests that match each duration on the y-axis.

This histogram shows a service that completes most requests in less than 300 ms. A small percentage of requests take up to 2 seconds, and a few outliers take more time.



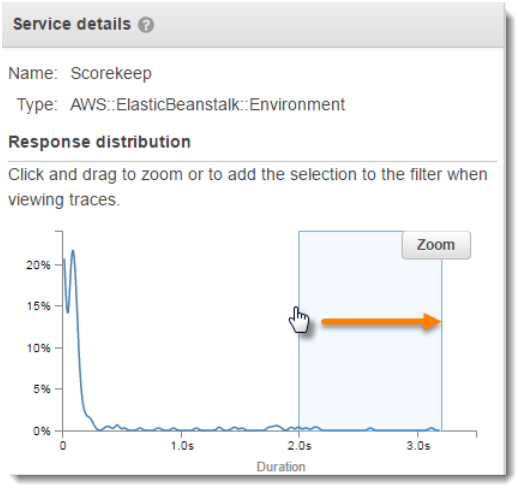

# Interpreting service details

Service histograms and edge histograms provide a visual representation of latency from the viewpoint of a service or requester.

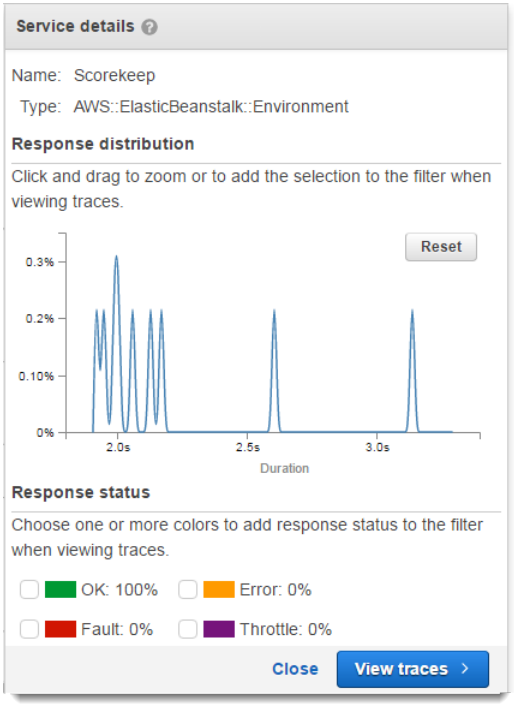
- Choose a *service node* by clicking the circle. X-Ray shows a histogram for requests served by the service. The latencies are those recorded by the service, and don't include any network latency between the service and the requester.
- Choose an *edge* by clicking the line or arrow tip of the edge between two services. X-Ray shows a histogram for requests from the requester that were served by the downstream service. The latencies are those recorded by the requester, and include latency in the network connection between the two services.

To interpret the **Service details** panel histogram, you can look for values that differ the most from the majority of values in the histogram. These *outliers* can be seen as peaks or spikes in the histogram, and you can view the traces for a specific area to investigate what's going on.

To view traces filtered by latency, select a range on the histogram. Click where you want to start the selection and drag from left to right to highlight a range of latencies to include in the trace filter.



After selecting a range, you can choose **Zoom** to view just that portion of the histogram and refine your selection.




Once you have the focus set to the area you're interested in, choose **View traces**.

# Configuring sampling rules in the X-Ray console

You can use the AWS X-Ray console to configure sampling rules for your services. The X-Ray SDK and AWS services that support active tracing (p. 6) with sampling configuration use sampling rules to determine which requests to record.

**Topics**



## Configuring sampling rules

You can configure sampling for the following use cases:

- **API Gateway Entrypoint** – API Gateway supports sampling and active tracing. To enable active tracing on an API stage, see Amazon API Gateway active tracing support for AWS X-Ray (p. 153).
- **AWS AppSync** – AWS AppSync supports sampling and active tracing. To enable active tracing on AWS AppSync requests, see Tracing with AWS X-Ray.

- **Instrument X-Ray SDK on compute platforms** – When using compute platforms such as Amazon EC2, Amazon ECS, or AWS Elastic Beanstalk, sampling is supported when the application has been instrumented with the lastest X-Ray SDK.

# Customizing sampling rules

By customizing sampling rules, you can control the amount of data that you record, and modify sampling behavior on the fly without modifying or redeploying your code. Sampling rules tell the X-Ray SDK how many requests to record for a set of criteria. By default, the X-Ray SDK records the first request each second, and five percent of any additional requests. One request per second is the *reservoir*. This ensures that at least one trace is recorded each second as long as the service is serving requests. Five percent is the *rate* at which additional requests beyond the reservoir size are sampled.

You can configure the X-Ray SDK to read sampling rules from a JSON document that you include with your code. However, when you run multiple instances of your service, each instance performs sampling independently. This causes the overall percentage of requests sampled to increase because the reservoirs of all of the instances are effectively added together. Additionally, to update local sampling rules, you need to redeploy your code.

By defining sampling rules in the X-Ray console, and configuring the SDK (p. 73) to read rules from the X-Ray service, you can avoid both of these issues. The service manages the reservoir for each rule, and assigns quotas to each instance of your service to distribute the reservoir evenly, based on the number of instances that are running. The reservoir limit is calculated according to the rules you set. And because the rules are configured in the service, you can manage rules without making additional deployments.

### To configure sampling rules in the X-Ray console

1. Open the X-Ray console.
2. Choose **Sampling**.
3. To create a rule, choose **Create sampling rule**.

   To edit a rule, choose a rule's name.

   To delete a rule, choose a rule and use the **Actions** menu to delete it.

# Sampling rule options

The following options are available for each rule. String values can use wildcards to match a single character (`?`) or zero or more characters (`*`).

### Sampling rule options

- **Rule name** (string) – A unique name for the rule.
- **Priority** (integer between 1 and 9999) – The priority of the sampling rule. Services evaluate rules in ascending order of priority, and make a sampling decision with the first rule that matches.
- **Reservoir** (non-negative integer) – A fixed number of matching requests to instrument per second, before applying the fixed rate. The reservoir is not used directly by services, but applies to all services using the rule collectively.
- **Rate** (number between 0 and 100) – The percentage of matching requests to instrument, after the reservoir is exhausted.
- **Service name** (string) – The name of the instrumented service, as it appears in the service map.
  - X-Ray SDK – The service name that you configure on the recorder.
  - Amazon API Gateway – *`api-name`*`/`*`stage`*.

- **Service type** (string) – The service type, as it appears in the service map. For the X-Ray SDK, set the service type by applying the appropriate plugin:
  - `AWS::ElasticBeanstalk::Environment` – An AWS Elastic Beanstalk environment (plugin).
  - `AWS::EC2::Instance` – An Amazon EC2 instance (plugin).
  - `AWS::ECS::Container` – An Amazon ECS container (plugin).
  - `AWS::APIGateway::Stage` – An Amazon API Gateway stage.
  - `AWS::AppSync::GraphQLAPI` – An AWS AppSync API request.
- **Host** (string) – The hostname from the HTTP host header.
- **HTTP method** (string) – The method of the HTTP request.
- **URL path** (string) – The URL path of the request.
  - X-Ray SDK – The path portion of the HTTP request URL.
  - Amazon API Gateway – Not supported.
- **Resource ARN** (string) – The ARN of the AWS resource running the service.
  - X-Ray SDK – Not supported. The SDK can only use rules with **Resource ARN** set to `*`.
  - Amazon API Gateway – The stage ARN.
- (Optional) **Attributes** (key and value) – Segment attributes that are known when the sampling decision is made.
  - X-Ray SDK – Not supported. The SDK ignores rules that specify attributes.
  - Amazon API Gateway – Headers from the original HTTP request.

# Sampling rule examples

### Example – Default rule with no reservoir and a low rate

You can modify the default rule's reservoir and rate. The default rule applies to requests that don't match any other rule.

- **Reservoir** – **`0`**
- **Rate** – **`0.005`** (0.5 percent)

### Example – Debugging rule to trace all requests for a problematic route

A high-priority rule applied temporarily for debugging.

- **Rule name** – **`DEBUG - history updates`**
- **Priority** – **`1`**
- **Reservoir** – **`1`**
- **Rate** – **`1`**
- **Service name** – **`Scorekeep`**
- **Service type** – **`*`**
- **Host** – **`*`**
- **HTTP method** – **`PUT`**
- **URL path** – **`/history/*`**
- **Resource ARN** – **`*`**

### Example – Higher minimum rate for POSTs

- **Rule name** – **`POST minimum`**

- **Priority** – `100`
- **Reservoir** – `10`
- **Rate** – `0.10`
- **Service name** – `*`
- **Service type** – `*`
- **Host** – `*`
- **HTTP method** – `POST`
- **URL path** – `*`
- **Resource ARN** – `*`

# Configuring your service to use sampling rules

The X-Ray SDK requires additional configuration to use sampling rules that you configure in the console. See the configuration topic for your language for details on configuring a sampling strategy:

- Java – Sampling rules (p. 191)
- Go – Sampling rules (p. 177)
- Node.js – Sampling rules (p. 217)
- Python – Sampling rules (p. 234)
- Ruby – Sampling rules (p. 255)
- .NET – Sampling rules (p. 269)

For API Gateway, see Amazon API Gateway active tracing support for AWS X-Ray (p. 153).

# Viewing sampling results

The X-Ray console **Sampling** page shows detailed information about how your services use each sampling rule.

The **Trend** column shows how the rule has been used in the last few minutes. Each column shows statistics for a 10-second window.

### Sampling statistics

- **Total matched rule** – The number of requests that matched this rule. This number doesn't include requests that could have matched this rule, but matched a higher-priority rule first.
- **Total sampled** – The number of requests recorded.
- **Sampled with fixed rate** – The number of requests sampled by applying the rule's fixed rate.
- **Sampled with reservoir limit** – The number of requests sampled using a quota assigned by X-Ray.
- **Borrowed from reservoir** – The number of requests sampled by borrowing from the reservoir. The first time a service matches a request to a rule, it has not yet been assigned a quota by X-Ray. However, if the reservoir is at least 1, the service borrows one trace per second until X-Ray assigns a quota.

For more information about sampling statistics and how services use sampling rules, see Using sampling rules with the X-Ray API (p. 100).

# Next steps

You can use the X-Ray API to manage sampling rules. With the API, you can create and update rules programmatically on a schedule, or in response to alarms or notifications. See Configuring sampling,

groups, and encryption settings with the AWS X-Ray API (p. 95) for instructions and additional rule examples.

The X-Ray SDK and AWS services also use the X-Ray API to read sampling rules, report sampling results, and get sampling targets. Services must keep track of how often they apply each rule, evaluate rules based on priority, and borrow from the reservoir when a request matches a rule for which X-Ray has not yet assigned the service a quota. For more detail about how a service uses the API for sampling, see Using sampling rules with the X-Ray API (p. 100).

When the X-Ray SDK calls sampling APIs, it uses the X-Ray daemon as a proxy. If you already use TCP port 2000, you can configure the daemon to run the proxy on a different port. See Configuring the AWS X-Ray daemon (p. 140) for details.

# Interacting with the AWS X-Ray Analytics console

The AWS X-Ray Analytics console is an interactive tool for interpreting trace data to quickly understand how your application and its underlying services are performing. The console enables you to explore, analyze, and visualize traces through interactive response time and time-series graphs.

When making selections in the Analytics console, the console constructs filters to reflect the selected subset of all traces. You can refine the active dataset with increasingly granular filters by clicking the graphs and the panels of metrics and fields that are associated with the current trace set.

**Topics**



## Console features

The X-Ray Analytics console uses the following key features for grouping, filtering, comparing, and quantifying trace data.

### Features

| Feature | Description |
| --- | --- |
| **Groups** | The initial selected group is `Default`. To change the retrieved group, select a different group from the menu to the right of the main filter expression search bar. To learn more about groups see, Using filter expressions with groups. |
| **Retrieved traces** | By default, the Analytics console generates graphs based on all traces in the selected group. |

| Feature | Description |
| --- | --- |
| | Retrieved traces represent all traces in your working set. You can find the trace count in this tile. Filter expressions you apply to the main search bar refine and update the retrieved traces. |
| **Show in charts/Hide from charts** | A toggle to compare the active group against the retrieved traces. To compare the data related to the group against any active filters, choose **Show in charts**. To remove this view from the charts, choose **Hide from charts**. |
| **Filtered trace set A** | Through interactions with the graphs and tables, apply filters to create the criteria for **Filtered trace set A**. As the filters are applied, the number of applicable traces and the percentage of traces from the total that are retrieved are calculated within this tile. Filters populate as tags within the **Filtered trace set A** tile and can also be removed from the tile. |
| **Refine** | This function updates the set of retrieved traces based on the filters applied to trace set A. Refining the retrieved trace set refreshes the working set of all traces retrieved based on the filters for trace set A. The working set of retrieved traces is a sampled subset of all traces in the group. |
| **Filtered trace set B** | When created, **Filtered trace set B** is a copy of **Filtered trace set A**. To compare the two trace sets, make new filter selections that will apply to trace set B, while trace set A remains fixed. As the filters are applied, the number of applicable traces and the percentage of traces from the total retrieved are calculated within this tile. Filters populate as tags within the **Filtered trace set B** tile and can also be removed from the tile. |
| **Response time root cause entity paths** | A table of recorded entity paths. X-Ray determines which path in your trace is the most likely cause for the response time. The format indicates a hierarchy of entities that are encountered, ending in a response time root cause. Use these rows to filter for recurring response time faults. For more information about customizing a root cause filter and getting data through the API see, Retrieving and refining root cause analytics. |
| **Delta (&#xFFFD;)** | A column that is added to the metrics tables when both trace set A and trace set B are active. The Delta column calculates the difference in percentage of traces between trace set A and trace set B. |

# Response time distribution

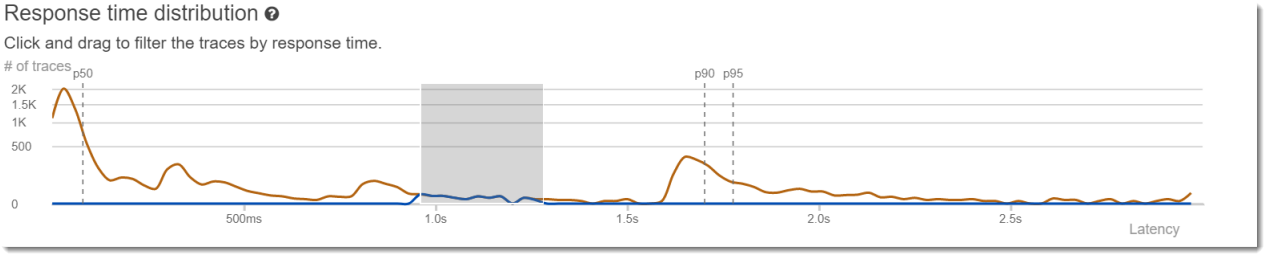
The X-Ray Analytics console generates two primary graphs to help you visualize traces: **Response Time Distribution** and **Time Series Activity**. This section and the following provide examples of each, and explain the basics of how to read the graphs.

The following are the colors associated with the response time line graph (the time series graph uses the same color scheme):

- **All traces in the group** – gray
- **Retrieved traces** – orange
- **Filtered trace set A** – green
- **Filtered trace set B** – blue

### Example – Response time distribution

The response time distribution is a chart that shows the number of traces with a given response time. Click and drag to make selections within the response time distribution. This selects and creates a filter on the working trace set named `responseTime` for all traces within a specific response time.

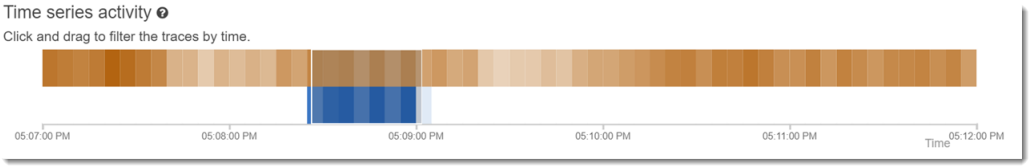



# Time series activity

The time series activity chart shows the number of traces at a given time period. The color indicators mirror the line graph colors of the response time distribution. The darker and fuller the color block within the activity series, the more traces are represented at the given time.

### Example – Time series activity

Click and drag to make selections within the time series activity graph. This selects and creates a filter named `timerange` on the working trace set for all traces within a specific range of time.



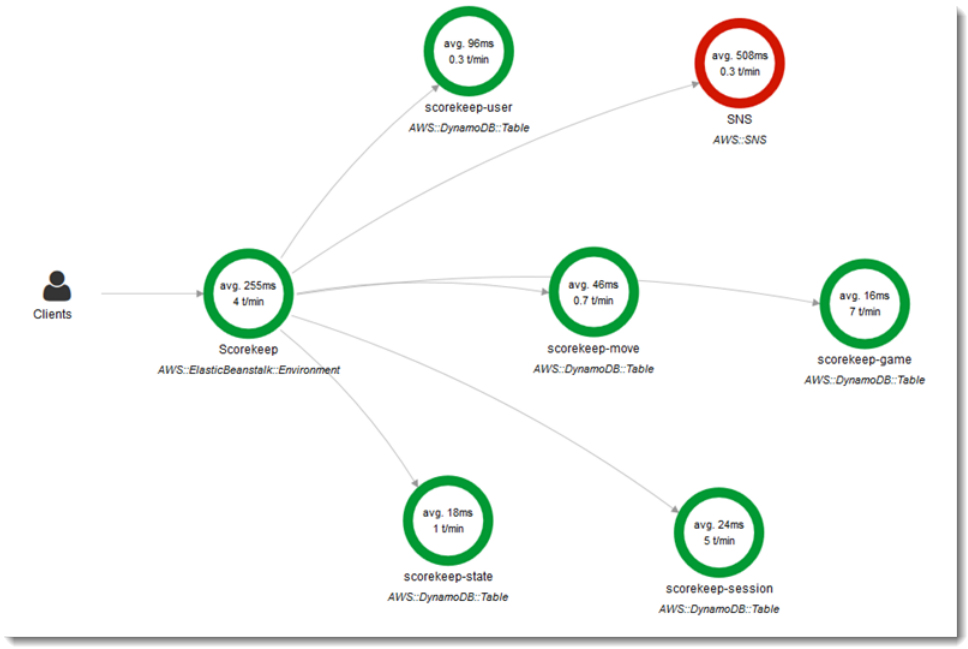

# Workflow examples

The following examples show common use cases for the X-Ray Analytics console. Each example demonstrates a key function of the console experience. As a group, the examples follow a basic troubleshooting workflow. The steps walk through how to first spot unhealthy nodes, and then how to interact with the Analytics console to automatically generate comparative queries. Once you have narrowed the scope through queries, you will finally look at the details of traces of interest to determine what is damaging the health of your service.

# Observe faults on the service graph

The service map indicates the health of each node by coloring it based on the ratio of successful calls to errors and faults. When you see a percentage of red on your node, it signals a fault. Use the X-Ray Analytics console to investigate it.

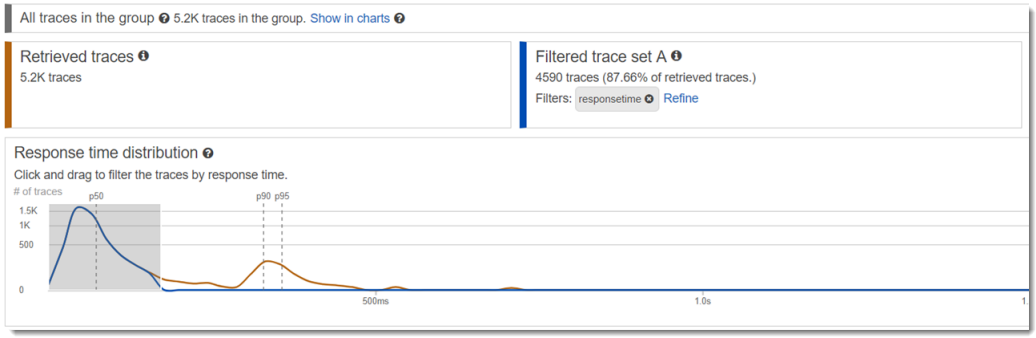
For more information about how to read the service map, see Viewing the service map.




# Identify response time peaks

Using the response time distribution, you can observe peaks in response time. By selecting the peak in response time, the tables below the graphs will update to expose all associated metrics, such as status codes.

When you click and drag, X-Ray selects and creates a filter. It's shown in a gray shadow on top of the graphed lines. You can now drag that shadow left and right along the distribution to update your selection and filter.




# View all traces marked with a status code

You can drill into traces within the selected peak by using the metrics tables below the graphs. By clicking a row in the **HTTP STATUS CODE** table, you automatically create a filter on the working dataset.

For example, you could view all traces of status code 500. This creates a filter tag in the trace set tile named `http.status`.

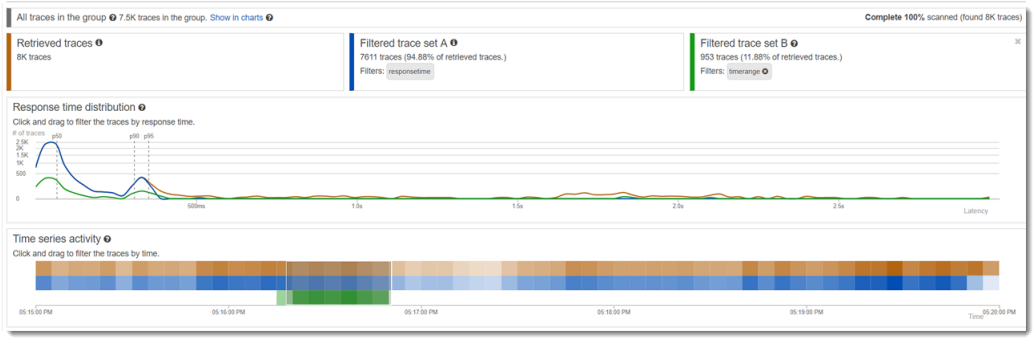
# View all items in a subgroup and associated to a user

Drill into the error set based on user, URL, response time root cause, or other predefined attributes. For example, to additionally filter the set of traces with a 500 status code, select a row from the **USERS** table. This results in two filter tags in the trace set tile: `http.status`, as designated previously, and `user`.

# Compare two sets of traces with different criteria

Compare across various users and their POST requests to find other discrepancies and correlations. Apply your first set of filters. They are defined by a blue line in the response time distribution. Then select **Compare**. Initially, this creates a copy of the filters on trace set A.

To proceed, define a new set of filters to apply to trace set B. This second set is represented by a green line. The following example shows different lines according to the blue and green color scheme.




# Identify a trace of interest and view its details

As you narrow your scope using the console filters, the trace list below the metrics tables becomes more meaningful. The trace list table combines information about **URL**, **USER**, and **STATUS CODE** into one view. For more insights, select a row from this table to open the trace's detail page and view its timeline and raw data.

# AWS X-Ray API

The X-Ray API provides access to all X-Ray functionality through the AWS SDK, AWS Command Line Interface, or directly over HTTPS. The X-Ray API Reference documents input parameters each API action, and the fields and data types that they return.

You can use the AWS SDK to develop programs that use the X-Ray API. The X-Ray console and X-Ray daemon both use the AWS SDK to communicate with X-Ray. The AWS SDK for each language has a reference document for classes and methods that map to X-Ray API actions and types.

**AWS SDK References**

- **Java** – AWS SDK for Java
- **JavaScript** – AWS SDK for JavaScript
- **.NET** – AWS SDK for .NET
- **Ruby** – AWS SDK for Ruby
- **Go** – AWS SDK for Go
- **PHP** – AWS SDK for PHP
- **Python** – AWS SDK for Python (Boto)

The AWS Command Line Interface is a command line tool that uses the SDK for Python to call AWS APIs. When you are first learning an AWS API, the AWS CLI provides an easy way to explore the available parameters and view the service output in JSON or text form.

See the AWS CLI Command Reference for details on `aws xray` subcommands.

**Topics**



# Using the AWS X-Ray API with the AWS CLI

The AWS CLI lets your access the X-Ray service directly and use the same APIs that the X-Ray console uses to retrieve the service graph and raw traces data. The sample application includes scripts that show how to use these APIs with the AWS CLI.

## Prerequisites

This tutorial uses the Scorekeep sample application and included scripts to generate tracing data and a service map. Follow the instructions in the getting started tutorial (p. 8) to launch the application.

This tutorial uses the AWS CLI to show basic use of the X-Ray API. The AWS CLI, available for Windows, Linux, and OS-X, provides command line access to the public APIs for all AWS services.

**Note**
You must verify that your AWS CLI is configured to the same Region that your Scorekeep sample application was created in.

Scripts included to test the sample application uses `cURL` to send traffic to the API and `jq` to parse the output. You can download the `jq` executable from stedolan.github.io, and the `curl` executable from https://curl.haxx.se/download.html. Most Linux and OS X installations include cURL.

# Generate trace data

The web app continues to generate traffic to the API every few seconds while the game is in-progress, but only generates one type of request. Use the `test-api.sh` script to run end to end scenarios and generate more diverse trace data while you test the API.

**To use the `test-api.sh` script**

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Copy the environment **URL** from the page header.
4. Open `bin/test-api.sh` and replace the value for API with your environment's URL.

```
#!/bin/bash
API=scorekeep.9hbtbm23t2.us-west-2.elasticbeanstalk.com/api
```

5. Run the script to generate traffic to the API.

```
~/debugger-tutorial$ ./bin/test-api.sh
Creating users,
session,
game,
configuring game,
playing game,
ending game,
game complete.
{"id":"MTBP8BAS","session":"HUF6IT64","name":"tic-tac-toe-test","users":
["QFF3HBGM","KL6JR98D"],"rules":"102","startTime":1476314241,"endTime":1476314245,"states":
["JQVLEOM2","D67QLPIC","VF9BM9NC","OEAA6GK9","2A705O73","1U2LFTLJ","HUKIDD70","BAN1C8FI","G3UDJTUF"
["BS8F8LQ","4MTTSPKP","463OETES","SVEBCL3N","N7CQ1GHP","O84ONEPD","EG4BPROQ","V4BLIDJ3","9RL3NPMV"]
```

# Use the X-Ray API

The AWS CLI provides commands for all of the API actions that X-Ray provides, including `GetServiceGraph` and `GetTraceSummaries`. See the AWS X-Ray API Reference for more information on all of the supported actions and the data types that they use.

**Example bin/service-graph.sh**

```
EPOCH=$(date +%s)
aws xray get-service-graph --start-time $(($EPOCH-600)) --end-time $EPOCH
```

The script retrieves a service graph for the last 10 minutes.

```
~/eb-java-scorekeep$ ./bin/service-graph.sh | less
{
    "StartTime": 1479068648.0,
    "Services": [
```

```
        {
            "StartTime": 1479068648.0,
            "ReferenceId": 0,
            "State": "unknown",
            "EndTime": 1479068651.0,
            "Type": "client",
            "Edges": [
                {
                    "StartTime": 1479068648.0,
                    "ReferenceId": 1,
                    "SummaryStatistics": {
                        "ErrorStatistics": {
                            "ThrottleCount": 0,
                            "TotalCount": 0,
                            "OtherCount": 0
                        },
                        "FaultStatistics": {
                            "TotalCount": 0,
                            "OtherCount": 0
                        },
                        "TotalCount": 2,
                        "OkCount": 2,
                        "TotalResponseTime": 0.054000139236450195
                    },
                    "EndTime": 1479068651.0,
                    "Aliases": []
                }
            ]
        },
        {
            "StartTime": 1479068648.0,
            "Names": [
                "scorekeep.elasticbeanstalk.com"
            ],
            "ReferenceId": 1,
            "State": "active",
            "EndTime": 1479068651.0,
            "Root": true,
            "Name": "scorekeep.elasticbeanstalk.com",
...
```

### Example bin/trace-urls.sh

```
EPOCH=$(date +%s)
aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time $(($EPOCH-60)) --query
 'TraceSummaries[*].Http.HttpURL'
```

The script retrieves the URLs of traces generated between one and two minutes ago.

```
~/eb-java-scorekeep$ ./bin/trace-urls.sh
[
    "http://scorekeep.elasticbeanstalk.com/api/game/6Q0UE1DG/5FGLM9U3/endtime/1479069438",
    "http://scorekeep.elasticbeanstalk.com/api/session/KH4341QH",
    "http://scorekeep.elasticbeanstalk.com/api/game/GLQBJ3K5/153AHDIA",
    "http://scorekeep.elasticbeanstalk.com/api/game/VPDL672J/G2V41HM6/endtime/1479069466"
]
```

### Example bin/full-traces.sh

```
EPOCH=$(date +%s)
TRACEIDS=$(aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time
 $(($EPOCH-60)) --query 'TraceSummaries[*].Id' --output text)
```

```
aws xray batch-get-traces --trace-ids $TRACEIDS --query 'Traces[*]'
```

The script retrieves full traces generated between one and two minutes ago.

```
~/eb-java-scorekeep$ ./bin/full-traces.sh | less
[
    {
        "Segments": [
            {
                "Id": "3f212bc237bafd5d",
                "Document": "{\"id\":\"3f212bc237bafd5d\",\"name\":\"DynamoDB\",
\"trace_id\":\"1-5828d9f2-a90669393f4343211bc1cf75\",\"start_time\":1.479072242459E9,
\"end_time\":1.479072242477E9,\"parent_id\":\"72a08dcf87991ca9\",\"http\":{\"response
\":{\"content_length\":60,\"status\":200}},\"inferred\":true,\"aws\":{\"consistent_read
\":false,\"table_name\":\"scorekeep-session-xray\",\"operation\":\"GetItem\",\"request_id
\":\"QAKE0S8DD0LJM245KAOPMA746BVV4KQNSO5AEMVJF66Q9ASUAAJG\",\"resource_names\":
[\"scorekeep-session-xray\"]},\"origin\":\"AWS::DynamoDB::Table\"}"
            },
            {
                "Id": "309e355f1148347f",
                "Document": "{\"id\":\"309e355f1148347f\",\"name\":\"DynamoDB\",
\"trace_id\":\"1-5828d9f2-a90669393f4343211bc1cf75\",\"start_time\":1.479072242477E9,
\"end_time\":1.479072242494E9,\"parent_id\":\"37f14ef837f00022\",\"http\":
{\"response\":{\"content_length\":606,\"status\":200}},\"inferred\":true,\"aws\":
{\"table_name\":\"scorekeep-game-xray\",\"operation\":\"UpdateItem\",\"request_id\":
\"388GEROC4PCA6D59ED3CTI5EEJVV4KQNSO5AEMVJF66Q9ASUAAJG\",\"resource_names\":[\"scorekeep-
game-xray\"]},\"origin\":\"AWS::DynamoDB::Table\"}"
            }
        ],
        "Id": "1-5828d9f2-a90669393f4343211bc1cf75",
        "Duration": 0.05099987983703613
    }
...
```

# Cleanup

Terminate your Elastic Beanstalk environment to shut down the Amazon EC2 instances, DynamoDB tables and other resources.

**To terminate your Elastic Beanstalk environment**

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Choose **Actions**.
4. Choose **Terminate Environment**.
5. Choose **Terminate**.

Trace data is automatically deleted from X-Ray after 30 days.

# Sending trace data to AWS X-Ray

You can send trace data to X-Ray in the form of segment documents. A segment document is a JSON formatted string that contains information about the work that your application does in service of a request. Your application can record data about the work that it does itself in segments, or work that uses downstream services and resources in subsegments.

Segments record information about the work that your application does. A segment, at a minimum, records the time spent on a task, a name, and two IDs. The trace ID tracks the request as it travels between services. The segment ID tracks the work done for the request by a single service.

### Example Minimal complete segment

```
{
  "name" : "Scorekeep",
  "id" : "70de5b6f19ff9a0a",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "end_time" : 1.478293361449E9
}
```

When a request is received, you can send an in-progress segment as a placeholder until the request is completed.

### Example In-progress segment

```
{
  "name" : "Scorekeep",
  "id" : "70de5b6f19ff9a0b",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  “in_progress”: true
}
```

You can send segments to X-Ray directly, with `PutTraceSegments` (p. 84), or through the X-Ray daemon (p. 85).

Most applications call other services or access resources with the AWS SDK. Record information about downstream calls in *subsegments*. X-Ray uses subsegments to identify downstream services that don't send segments and create entries for them on the service graph.

A subsegment can be embedded in a full segment document, or sent separately. Send subsegments separately to asynchronously trace downstream calls for long-running requests, or to avoid exceeding the maximum segment document size (64 kB).

### Example Subsegment

A subsegment has a `type` of `subsegment` and a `parent_id` that identifies the parent segment.

```
{
  "name" : "www2.example.com",
  "id" : "70de5b6f19ff9a0c",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979"
  “end_time” : 1.478293361449E9,
  “type” : “subsegment”,
  “parent_id” : “70de5b6f19ff9a0b”
}
```

For more information on the fields and values that you can include in segments and subsegments, see AWS X-Ray segment documents (p. 103).

**Sections**

# Generating trace IDs

To send data to X-Ray, you need to generate a unique trace ID for each request. Trace IDs must meet the following requirements.

### Trace ID Format

A `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

- The version number, that is, `1`.
- The time of the original request, in Unix epoch time, in **8 hexadecimal digits**.

  For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal digits.
- A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.

You can write a script to generate trace IDs for testing. Here are two examples.

**Python**

```
import time
import os
import binascii

START_TIME = time.time()
HEX=hex(int(START_TIME))[2:]
TRACE_ID="1-{}-{}".format(HEX, binascii.hexlify(os.urandom(12)))
```

**Bash**

```
START_TIME=$(date +%s)
HEX_TIME=$(printf '%x\n' $START_TIME)
GUID=$(dd if=/dev/random bs=12 count=1 2>/dev/null | od -An -tx1 | tr -d ' \t\n')
TRACE_ID="1-$HEX_TIME-$GUID"
```

See the Scorekeep sample application for scripts that create trace IDs and send segments to the X-Ray daemon.

- Python – `xray_start.py`
- Bash – `xray_start.sh`

# Using PutTraceSegments

You can upload segment documents with the `PutTraceSegments` API. The API has a single parameter, `TraceSegmentDocuments`, that takes a list of JSON segment documents.

With the AWS CLI, use the `aws xray put-trace-segments` command to send segment documents directly to X-Ray.

```
$ DOC='{"trace_id": "1-5960082b-ab52431b496add878434aa25", "id": "6226467e3f845502",
 "start_time": 1498082657.37518, "end_time": 1498082695.4042, "name":
 "test.elasticbeanstalk.com"}'
$ aws xray put-trace-segments --trace-segment-documents "$DOC"
{
```

```
    "UnprocessedTraceSegments": []
}
```

**Note**
Windows Command Processor and Windows PowerShell have different requirements for quoting and escaping quotes in JSON strings. See Quoting Strings in the AWS CLI User Guide for details.

The output lists any segments that failed processing. For example, if the date in the trace ID is too far in the past, you see an error like the following.

```
{
    "UnprocessedTraceSegments": [
        {
            "ErrorCode": "InvalidTraceId",
            "Message": "Invalid segment. ErrorCode: InvalidTraceId",
            "Id": "6226467e3f845502"
        }
    ]
}
```

You can pass multiple segment documents at the same time, separated by spaces.

```
$ aws xray put-trace-segments --trace-segment-documents "$DOC1" "$DOC2"
```

# Sending segment documents to the X-Ray daemon

Instead of sending segment documents to the X-Ray API, you can send segments and subsegments to the X-Ray daemon, which will buffer them and upload to the X-Ray API in batches. The X-Ray SDK sends segment documents to the daemon to avoid making calls to AWS directly.

**Note**
See Running the X-Ray daemon locally (p. 142) for instructions on running the daemon.

Send the segment in JSON over UDP port 2000, prepended by the daemon header, `{"format": "json", "version": 1}\n`

```
{"format": "json", "version": 1}\n{"trace_id": "1-5759e988-bd862e3fe1be46a994272793", "id":
 "defdfd9912dc5a56", "start_time": 1461096053.37518, "end_time": 1461096053.4042, "name":
 "test.elasticbeanstalk.com"}
```

On Linux, you can send segment documents to the daemon from a Bash terminal. Save the header and segment document to a text file and pipe it to `/dev/udp` with `cat`.

```
$ cat segment.txt > /dev/udp/127.0.0.1/2000
```

**Example segment.txt**

```
{"format": "json", "version": 1}
{"trace_id": "1-594aed87-ad72e26896b3f9d3a27054bb", "id": "6226467e3f845502", "start_time":
 1498082657.37518, "end_time": 1498082695.4042, "name": "test.elasticbeanstalk.com"}
```

Check the daemon log (p. 139) to verify that it sent the segment to X-Ray.

```
2017-07-07T01:57:24Z [Debug] processor: sending partial batch
2017-07-07T01:57:24Z [Debug] processor: segment batch size: 1. capacity: 50
```

```
2017-07-07T01:57:24Z [Info] Successfully sent batch of 1 segments (0.020 seconds)
```

# Getting data from AWS X-Ray

AWS X-Ray processes the trace data that you send to it to generate full traces, trace summaries, and service graphs in JSON. You can retrieve the generated data directly from the API with the AWS CLI.

**Sections**



## Retrieving the service graph

You can use the `GetServiceGraph` API to retrieve the JSON service graph. The API requires a start time and end time, which you can calculate from a Linux terminal with the `date` command.

```
$ date +%s
1499394617
```

`date +%s` prints a date in seconds. Use this number as an end time and subtract time from it to get a start time.

**Example Script to retrieve a service graph for the last 10 minutes**

```
EPOCH=$(date +%s)
aws xray get-service-graph --start-time $(($EPOCH-600)) --end-time $EPOCH
```

The following example shows a service graph with 4 nodes, including a client node, an EC2 instance, a DynamoDB table, and an Amazon SNS topic.

**Example GetServiceGraph output**

```
{
    "Services": [
        {
            "ReferenceId": 0,
            "Name": "xray-sample.elasticbeanstalk.com",
            "Names": [
                "xray-sample.elasticbeanstalk.com"
            ],
            "Type": "client",
            "State": "unknown",
            "StartTime": 1528317567.0,
            "EndTime": 1528317589.0,
            "Edges": [
                {
                    "ReferenceId": 2,
                    "StartTime": 1528317567.0,
                    "EndTime": 1528317589.0,
                    "SummaryStatistics": {
                        "OkCount": 3,
                        "ErrorStatistics": {
```

```
                        "ThrottleCount": 0,
                        "OtherCount": 1,
                        "TotalCount": 1
                    },
                    "FaultStatistics": {
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
                    "TotalCount": 4,
                    "TotalResponseTime": 0.273
                },
                "ResponseTimeHistogram": [
                    {
                        "Value": 0.005,
                        "Count": 1
                    },
                    {
                        "Value": 0.015,
                        "Count": 1
                    },
                    {
                        "Value": 0.157,
                        "Count": 1
                    },
                    {
                        "Value": 0.096,
                        "Count": 1
                    }
                ],
                "Aliases": []
            }
        ]
    },
    {
        "ReferenceId": 1,
        "Name": "awseb-e-dixzws4s9p-stack-StartupSignupsTable-4IMSMHAYX2BA",
        "Names": [
            "awseb-e-dixzws4s9p-stack-StartupSignupsTable-4IMSMHAYX2BA"
        ],
        "Type": "AWS::DynamoDB::Table",
        "State": "unknown",
        "StartTime": 1528317583.0,
        "EndTime": 1528317589.0,
        "Edges": [],
        "SummaryStatistics": {
            "OkCount": 2,
            "ErrorStatistics": {
                "ThrottleCount": 0,
                "OtherCount": 0,
                "TotalCount": 0
            },
            "FaultStatistics": {
                "OtherCount": 0,
                "TotalCount": 0
            },
            "TotalCount": 2,
            "TotalResponseTime": 0.12
        },
        "DurationHistogram": [
            {
                "Value": 0.076,
                "Count": 1
            },
            {
                "Value": 0.044,
                "Count": 1
```

```
            }
        ],
        "ResponseTimeHistogram": [
            {
                "Value": 0.076,
                "Count": 1
            },
            {
                "Value": 0.044,
                "Count": 1
            }
        ]
    },
    {
        "ReferenceId": 2,
        "Name": "xray-sample.elasticbeanstalk.com",
        "Names": [
            "xray-sample.elasticbeanstalk.com"
        ],
        "Root": true,
        "Type": "AWS::EC2::Instance",
        "State": "active",
        "StartTime": 1528317567.0,
        "EndTime": 1528317589.0,
        "Edges": [
            {
                "ReferenceId": 1,
                "StartTime": 1528317567.0,
                "EndTime": 1528317589.0,
                "SummaryStatistics": {
                    "OkCount": 2,
                    "ErrorStatistics": {
                        "ThrottleCount": 0,
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
                    "FaultStatistics": {
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
                    "TotalCount": 2,
                    "TotalResponseTime": 0.12
                },
                "ResponseTimeHistogram": [
                    {
                        "Value": 0.076,
                        "Count": 1
                    },
                    {
                        "Value": 0.044,
                        "Count": 1
                    }
                ],
                "Aliases": []
            },
            {
                "ReferenceId": 3,
                "StartTime": 1528317567.0,
                "EndTime": 1528317589.0,
                "SummaryStatistics": {
                    "OkCount": 2,
                    "ErrorStatistics": {
                        "ThrottleCount": 0,
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
```

```
                    "FaultStatistics": {
                        "OtherCount": 0,
                        "TotalCount": 0
                    },
                    "TotalCount": 2,
                    "TotalResponseTime": 0.125
                },
                "ResponseTimeHistogram": [
                    {
                        "Value": 0.049,
                        "Count": 1
                    },
                    {
                        "Value": 0.076,
                        "Count": 1
                    }
                ],
                "Aliases": []
            }
        ],
        "SummaryStatistics": {
            "OkCount": 3,
            "ErrorStatistics": {
                "ThrottleCount": 0,
                "OtherCount": 1,
                "TotalCount": 1
            },
            "FaultStatistics": {
                "OtherCount": 0,
                "TotalCount": 0
            },
            "TotalCount": 4,
            "TotalResponseTime": 0.273
        },
        "DurationHistogram": [
            {
                "Value": 0.005,
                "Count": 1
            },
            {
                "Value": 0.015,
                "Count": 1
            },
            {
                "Value": 0.157,
                "Count": 1
            },
            {
                "Value": 0.096,
                "Count": 1
            }
        ],
        "ResponseTimeHistogram": [
            {
                "Value": 0.005,
                "Count": 1
            },
            {
                "Value": 0.015,
                "Count": 1
            },
            {
                "Value": 0.157,
                "Count": 1
            },
            {
```

```
                    "Value": 0.096,
                    "Count": 1
                }
            ]
        },
        {
            "ReferenceId": 3,
            "Name": "SNS",
            "Names": [
                "SNS"
            ],
            "Type": "AWS::SNS",
            "State": "unknown",
            "StartTime": 1528317583.0,
            "EndTime": 1528317589.0,
            "Edges": [],
            "SummaryStatistics": {
                "OkCount": 2,
                "ErrorStatistics": {
                    "ThrottleCount": 0,
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "FaultStatistics": {
                    "OtherCount": 0,
                    "TotalCount": 0
                },
                "TotalCount": 2,
                "TotalResponseTime": 0.125
            },
            "DurationHistogram": [
                {
                    "Value": 0.049,
                    "Count": 1
                },
                {
                    "Value": 0.076,
                    "Count": 1
                }
            ],
            "ResponseTimeHistogram": [
                {
                    "Value": 0.049,
                    "Count": 1
                },
                {
                    "Value": 0.076,
                    "Count": 1
                }
            ]
        }
    ]
}
```


# Retrieving the service graph by group

To call for a service graph based on the contents of a group, include a `groupName` or `groupARN`. The following example shows a service graph call to a group named Example1.

**Example Script to retrieve a service graph by name for group Example1**

```
aws xray get-service-graph --group-name "Example1"
```

# Retrieving traces

You can use the `GetTraceSummaries` API to get a list of trace summaries. Trace summaries include information that you can use to identify traces that you want to download in full, including annotations, request and response information, and IDs.

There are two `TimeRangeType` flags available when calling `aws xray get-trace-summaries`:

- **TraceId** – The default `GetTraceSummaries` search uses TraceID time and returns traces started within the computed `[start_time, end_time)` range. This range of timestamps is calculated based on the encoding of the timestamp within the TraceId, or can be defined manually.
- **Event time** – To search for events as they happen over the time, AWS X-Ray allows searching for traces using event timestamps. Event time returns traces active during the `[start_time, end_time)` range, regardless of when the trace began.

Use the `aws xray get-trace-summaries` command to get a list of trace summaries. The following commands get a list of trace summaries from between 1 and 2 minutes in the past using the default TraceId time.

### Example Script to get trace summaries

```
EPOCH=$(date +%s)
aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time $(($EPOCH-60))
```

### Example GetTraceSummaries output

```
{
    "TraceSummaries": [
        {
            "HasError": false,
            "Http": {
                "HttpStatus": 200,
                "ClientIp": "205.255.255.183",
                "HttpURL": "http://scorekeep.elasticbeanstalk.com/api/session",
                "UserAgent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
                "HttpMethod": "POST"
            },
            "Users": [],
            "HasFault": false,
            "Annotations": {},
            "ResponseTime": 0.084,
            "Duration": 0.084,
            "Id": "1-59602606-a43a1ac52fc7ee0eea12a82c",
            "HasThrottle": false
        },
        {
            "HasError": false,
            "Http": {
                "HttpStatus": 200,
                "ClientIp": "205.255.255.183",
                "HttpURL": "http://scorekeep.elasticbeanstalk.com/api/user",
                "UserAgent": "Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36",
                "HttpMethod": "POST"
            },
            "Users": [
                {
                    "UserName": "5M388M1E"
```

```
                }
            ],
            "HasFault": false,
            "Annotations": {
                "UserID": [
                    {
                        "AnnotationValue": {
                            "StringValue": "5M388M1E"
                        }
                    }
                ],
                "Name": [
                    {
                        "AnnotationValue": {
                            "StringValue": "Ola"
                        }
                    }
                ]
            },
            "ResponseTime": 3.232,
            "Duration": 3.232,
            "Id": "1-59602603-23fc5b688855d396af79b496",
            "HasThrottle": false
        }
    ],
    "ApproximateTime": 1499473304.0,
    "TracesProcessedCount": 2
}
```

Use the trace ID from the output to retrieve a full trace with the `BatchGetTraces` API.

### Example BatchGetTraces command

```
$ aws xray batch-get-traces --trace-ids 1-596025b4-7170afe49f7aa708b1dd4a6b
```

### Example BatchGetTraces output

```
{
    "Traces": [
        {
            "Duration": 3.232,
            "Segments": [
                {
                    "Document": "{\"id\":\"1fb07842d944e714\",\"name\":
\"random-name\",\"start_time\":1.499473411677E9,\"end_time\":1.499473414572E9,
\"parent_id\":\"0c544c1b1bbff948\",\"http\":{\"response\":{\"status\":200}},
\"aws\":{\"request_id\":\"ac086670-6373-11e7-a174-f31b3397f190\"},\"trace_id\":
\"1-59602603-23fc5b688855d396af79b496\",\"origin\":\"AWS::Lambda\",\"resource_arn\":
\"arn:aws:lambda:us-west-2:123456789012:function:random-name\"}",
                    "Id": "1fb07842d944e714"
                },
                {
                    "Document": "{\"id\":\"194fcc8747581230\",\"name\":\"Scorekeep
\",\"start_time\":1.499473411562E9,\"end_time\":1.499473414794E9,\"http\":{\"request
\":{\"url\":\"http://scorekeep.elasticbeanstalk.com/api/user\",\"method\":\"POST\",
\"user_agent\":\"Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML,
 like Gecko) Chrome/59.0.3071.115 Safari/537.36\",\"client_ip\":\"205.251.233.183\"},
\"response\":{\"status\":200}},\"aws\":{\"elastic_beanstalk\":{\"version_label\":\"app-
abb9-170708_002045\",\"deployment_id\":406,\"environment_name\":\"scorekeep-dev\"},
\"ec2\":{\"availability_zone\":\"us-west-2c\",\"instance_id\":\"i-0cd9e448944061b4a
\"},\"xray\":{\"sdk_version\":\"1.1.2\",\"sdk\":\"X-Ray for Java\"}},\"service
\":{},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"user\":\"5M388M1E
\",\"origin\":\"AWS::ElasticBeanstalk::Environment\",\"subsegments\":[{\"id\":
```

```
\"0c544c1b1bbff948\",\"name\":\"Lambda\",\"start_time\":1.499473411629E9,\"end_time
\":1.499473414572E9,\"http\":{\"response\":{\"status\":200,\"content_length\":14}},
\"aws\":{\"log_type\":\"None\",\"status_code\":200,\"function_name\":\"random-name
\",\"invocation_type\":\"RequestResponse\",\"operation\":\"Invoke\",\"request_id
\":\"ac086670-6373-11e7-a174-f31b3397f190\",\"resource_names\":[\"random-name\"]},
\"namespace\":\"aws\"},{\"id\":\"071684f2e555e571\",\"name\":\"## UserModel.saveUser
\",\"start_time\":1.499473414581E9,\"end_time\":1.499473414769E9,\"metadata\":{\"debug
\":{\"test\":\"Metadata string from UserModel.saveUser\"}},\"subsegments\":[{\"id\":
\"4cd3f10b76c624b4\",\"name\":\"DynamoDB\",\"start_time\":1.49947341469E9,\"end_time
\":1.499473414769E9,\"http\":{\"response\":{\"status\":200,\"content_length\":57}},
\"aws\":{\"table_name\":\"scorekeep-user\",\"operation\":\"UpdateItem\",\"request_id\":
\"MFQ8CGJ3JTDDVVVASUAAJGQ6NJ82F738BOB4KQNSO5AEMVJF66Q9\",\"resource_names\":[\"scorekeep-
user\"]},\"namespace\":\"aws\"}]}]}",
                    "Id": "194fcc8747581230"
                },
                {
                    "Document": "{\"id\":\"00f91aa01f4984fd\",\"name\":
\"random-name\",\"start_time\":1.49947341283E9,\"end_time\":1.49947341457E9,
\"parent_id\":\"1fb07842d944e714\",\"aws\":{\"function_arn\":\"arn:aws:lambda:us-
west-2:123456789012:function:random-name\",\"resource_names\":[\"random-name\"],
\"account_id\":\"123456789012\"},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",
\"origin\":\"AWS::Lambda::Function\",\"subsegments\":[{\"id\":\"e6d2fe619f827804\",
\"name\":\"annotations\",\"start_time\":1.499473413012E9,\"end_time\":1.499473413069E9,
\"annotations\":{\"UserID\":\"5M388M1E\",\"Name\":\"Ola\"}},{\"id\":\"b29b548af4d54a0f
\",\"name\":\"SNS\",\"start_time\":1.499473413112E9,\"end_time\":1.499473414071E9,
\"http\":{\"response\":{\"status\":200}},\"aws\":{\"operation\":\"Publish\",\"region
\":\"us-west-2\",\"request_id\":\"a2137970-f6fc-5029-83e8-28aadeb99198\",\"retries
\":0,\"topic_arn\":\"arn:aws:sns:us-west-2:123456789012:awseb-e-ruag3jyweb-stack-
NotificationTopic-6B829NT9V5O9\"},\"namespace\":\"aws\"},{\"id\":\"2279c0030c955e52\",
\"name\":\"Initialization\",\"start_time\":1.499473412064E9,\"end_time\":1.499473412819E9,
\"aws\":{\"function_arn\":\"arn:aws:lambda:us-west-2:123456789012:function:random-name
\"}}]}",
                    "Id": "00f91aa01f4984fd"
                },
                {
                    "Document": "{\"id\":\"17ba309b32c7fbaf\",\"name\":
\"DynamoDB\",\"start_time\":1.49947341469E9,\"end_time\":1.499473414769E9,
\"parent_id\":\"4cd3f10b76c624b4\",\"inferred\":true,\"http\":{\"response
\":{\"status\":200,\"content_length\":57}},\"aws\":{\"table_name
\":\"scorekeep-user\",\"operation\":\"UpdateItem\",\"request_id\":
\"MFQ8CGJ3JTDDVVVASUAAJGQ6NJ82F738BOB4KQNSO5AEMVJF66Q9\",\"resource_names\":
[\"scorekeep-user\"]},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",\"origin\":
\"AWS::DynamoDB::Table\"}",
                    "Id": "17ba309b32c7fbaf"
                },
                {
                    "Document": "{\"id\":\"1ee3c4a523f89ca5\",\"name\":\"SNS\",\"start_time
\":1.499473413112E9,\"end_time\":1.499473414071E9,\"parent_id\":\"b29b548af4d54a0f\",
\"inferred\":true,\"http\":{\"response\":{\"status\":200}},\"aws\":{\"operation\":\"Publish
\",\"region\":\"us-west-2\",\"request_id\":\"a2137970-f6fc-5029-83e8-28aadeb99198\",
\"retries\":0,\"topic_arn\":\"arn:aws:sns:us-west-2:123456789012:awseb-e-ruag3jyweb-stack-
NotificationTopic-6B829NT9V5O9\"},\"trace_id\":\"1-59602603-23fc5b688855d396af79b496\",
\"origin\":\"AWS::SNS\"}",
                    "Id": "1ee3c4a523f89ca5"
                }
            ],
            "Id": "1-59602603-23fc5b688855d396af79b496"
        }
    ],
    "UnprocessedTraceIds": []
}
```


The full trace includes a document for each segment, compiled from all of the segment documents received with the same trace ID. These documents don't represent the data as it was sent to X-Ray by your application. Instead, they represent the processed documents generated by the X-Ray service. X-

Ray creates the full trace document by compiling segment documents sent by your application, and removing data that doesn't comply with the segment document schema (p. 103).

X-Ray also creates *inferred segments* for downstream calls to services that don't send segments themselves. For example, when you call DynamoDB with an instrumented client, the X-Ray SDK records a subsegment with details about the call from its point of view. However, DynamoDB doesn't send a corresponding segment. X-Ray uses the information in the subsegment to create an inferred segment to represent the DynamoDB resource in the service map, and adds it to the trace document.

To get multiple traces from the API, you need a list of trace IDs, which you can extract from the output of `get-trace-summaries` with an AWS CLI query. Redirect the list to the intput of `batch-get-traces` to get full traces for a specific time period.

**Example Script to get full traces for a one minute period**

```
EPOCH=$(date +%s)
TRACEIDS=$(aws xray get-trace-summaries --start-time $(($EPOCH-120)) --end-time
 $(($EPOCH-60)) --query 'TraceSummaries[*].Id' --output text)
aws xray batch-get-traces --trace-ids $TRACEIDS --query 'Traces[*]'
```

# Retrieving and refining root cause analytics

Upon generating a trace summary with the GetTraceSummaries API , partial trace summaries can be reused in their JSON format to create a refined filter expression based upon root causes. See the examples below for a walkthrough of the refinement steps.

**Example Example GetTraceSummaries output - response time root cause section**

```
{
  "Services": [
    {
      "Name": "GetWeatherData",
      "Names": ["GetWeatherData"],
      "AccountId": 123456789012,
      "Type": null,
      "Inferred": false,
      "EntityPath": [
        {
          "Name": "GetWeatherData",
          "Coverage": 1.0,
          'Remote": false
        },
        {
          "Name": "get_temperature",
          "Coverage": 0.8,
          "Remote": false
        }
      ]
    },
    {
      "Name": "GetTemperature",
      "Names": ["GetTemperature"],
      "AccountId": 123456789012,
      "Type": null,
      "Inferred": false,
      "EntityPath": [
        {
          "Name": "GetTemperature",
          "Coverage": 0.7,
          "Remote": false
```

```
        }
      ]
    }
  ]
}
```

By editing and making omissions to the above output, this JSON can become a filter for matched root cause entities. For every field present in the JSON, any candidate match must be exact, or the trace will not be returned. Removed fields become wildcard values, a format which is compatible with the filter expression query structure.

**Example Reformatted response time root cause**

```
{
  "Services": [
    {
      "Name": "GetWeatherData",
      "EntityPath": [
        {
          "Name": "GetWeatherData"
        },
        {
          "Name": "get_temperature"
        }
      ]
    },
    {
      "Name": "GetTemperature",
      "EntityPath": [
        {
          "Name": "GetTemperature"
        }
      ]
    }
  ]
}
```

This JSON is then used as part of a filter expression through a call to `rootcause.json = #[{}]`. Refer to the Filter Expressions (p. 59) chapter for more details about querying with filter expressions.

**Example Example JSON filter**

```
rootcause.json = #[{ "Services": [ { "Name": "GetWeatherData", "EntityPath": [{ "Name":
 "GetWeatherData" }, { "Name": "get_temperature" } ] }, { "Name": "GetTemperature",
 "EntityPath": [ { "Name": "GetTemperature" } ] } ] }]
```

# Configuring sampling, groups, and encryption settings with the AWS X-Ray API

AWS X-Ray provides APIs for configuring sampling rules (p. 70), group rules, and encryption settings (p. 29).

**Sections**

- Encryption settings (p. 96)
- Sampling rules (p. 96)

- Groups (p. 99)

# Encryption settings

Use `PutEncryptionConfig` to specify an AWS Key Management Service (AWS KMS) customer master key (CMK) to use for encryption.

**Note**
X-Ray does not support asymmetric CMKs.

```
$ aws xray put-encryption-config --type KMS --key-id alias/aws/xray
{
    "EncryptionConfig": {
        "KeyId": "arn:aws:kms:us-east-2:123456789012:key/c234g4e8-39e9-4gb0-84e2-
b0ea215cbba5",
        "Status": "UPDATING",
        "Type": "KMS"
    }
}
```

For the key ID, you can use an alias (as shown in the example), a key ID, or an Amazon Resource Name (ARN).

Use `GetEncryptionConfig` to get the current configuration. When X-Ray finishes applying your settings, the status changes from `UPDATING` to `ACTIVE`.

```
$ aws xray get-encryption-config
{
    "EncryptionConfig": {
        "KeyId": "arn:aws:kms:us-east-2:123456789012:key/c234g4e8-39e9-4gb0-84e2-
b0ea215cbba5",
        "Status": "ACTIVE",
        "Type": "KMS"
    }
}
```

To stop using a CMK and use default encryption, set the encryption type to `NONE`.

```
$ aws xray put-encryption-config --type NONE
{
    "EncryptionConfig": {
        "Status": "UPDATING",
        "Type": "NONE"
    }
}
```

# Sampling rules

You can manage the sampling rules (p. 70) in your account with the X-Ray API.

Get all sampling rules with `GetSamplingRules`.

```
$ aws xray get-sampling-rules
{
    "SamplingRuleRecords": [
        {
```

```
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-2:123456789012:sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.05,
                "ReservoirSize": 1,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1529959993.0
        }
    ]
}
```

The default rule applies to all requests that don't match another rule. It is the lowest priority rule and cannot be deleted. You can, however, change the rate and reservoir size with `UpdateSamplingRule`.

**Example API input for `UpdateSamplingRule` – 10000-default.json**

```
{
    "SamplingRuleUpdate": {
        "RuleName": "Default",
        "FixedRate": 0.01,
        "ReservoirSize": 0
    }
}
```

The following example uses the previous file as input to change the default rule to one percent with no reservoir.

```
$ aws xray update-sampling-rule --cli-input-json file://1000-default.json
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-2:123456789012:sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.01,
                "ReservoirSize": 0,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1529959993.0
        },
```

Create additional sampling rules with `CreateSamplingRule`. When you create a rule, most of the rule fields are required. The following example creates two rules. This first rule sets a base rate for

the Scorekeep sample application. It matches all requests served by the API that don't match a higher priority rule.

**Example API input for `UpdateSamplingRule` – 9000-base-scorekeep.json**

```
{
    "SamplingRule": {
        "RuleName": "base-scorekeep",
        "ResourceARN": "*",
        "Priority": 9000,
        "FixedRate": 0.1,
        "ReservoirSize": 5,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "*",
        "URLPath": "*",
        "Version": 1
    }
}
```

The second rule also applies to Scorekeep, but it has a higher priority and is more specific. This rule sets a very low sampling rate for polling requests. These are GET requests made by the client every few seconds to check for changes to the game state.

**Example API input for `UpdateSamplingRule` – 5000-polling-scorekeep.json**

```
{
    "SamplingRule": {
        "RuleName": "polling-scorekeep",
        "ResourceARN": "*",
        "Priority": 5000,
        "FixedRate": 0.003,
        "ReservoirSize": 0,
        "ServiceName": "Scorekeep",
        "ServiceType": "*",
        "Host": "*",
        "HTTPMethod": "GET",
        "URLPath": "/api/state/*",
        "Version": 1
    }
}
```

```
$ aws xray create-sampling-rule --cli-input-json file://5000-polling-scorekeep.json
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "polling-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/polling-
scorekeep",
            "ResourceARN": "*",
            "Priority": 5000,
            "FixedRate": 0.003,
            "ReservoirSize": 0,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "GET",
            "URLPath": "/api/state/*",
            "Version": 1,
            "Attributes": {}
```

```
        },
        "CreatedAt": 1530574399.0,
        "ModifiedAt": 1530574399.0
    }
}
$ aws xray create-sampling-rule --cli-input-json file://9000-base-scorekeep.json
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "base-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/base-scorekeep",
            "ResourceARN": "*",
            "Priority": 9000,
            "FixedRate": 0.1,
            "ReservoirSize": 5,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "*",
            "URLPath": "*",
            "Version": 1,
            "Attributes": {}
        },
        "CreatedAt": 1530574410.0,
        "ModifiedAt": 1530574410.0
    }
}
```

To delete a sampling rule, use `DeleteSamplingRule`.

```
$ aws xray delete-sampling-rule --rule-name polling-scorekeep
{
    "SamplingRuleRecord": {
        "SamplingRule": {
            "RuleName": "polling-scorekeep",
            "RuleARN": "arn:aws:xray:us-east-1:123456789012:sampling-rule/polling-
scorekeep",
            "ResourceARN": "*",
            "Priority": 5000,
            "FixedRate": 0.003,
            "ReservoirSize": 0,
            "ServiceName": "Scorekeep",
            "ServiceType": "*",
            "Host": "*",
            "HTTPMethod": "GET",
            "URLPath": "/api/state/*",
            "Version": 1,
            "Attributes": {}
        },
        "CreatedAt": 1530574399.0,
        "ModifiedAt": 1530574399.0
    }
}
```

# Groups

You can use the X-Ray API to manage groups in your account. Groups are a collection of traces that are defined by a filter expression. You can use groups to generate additional service graphs and supply Amazon CloudWatch metrics. See Getting data from AWS X-Ray (p. 86) for more details about working with service graphs and metrics through the X-Ray API.

Create a group with `CreateGroup`.

```
$ aws xray create-group --group-name "TestGroup" --filter-expression "service(\"example.com
\") {fault}"
{
    "GroupName": "TestGroup",
    "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
    "FilterExpression": "service(\"example.com\") {fault OR error}"
}
```

Get all existing groups with `GetGroups`.

```
$ aws xray get-groups
{
    "Groups": [
        {
            "GroupName": "TestGroup",
            "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
            "FilterExpression": "service(\"example.com\") {fault OR error}"
        },
  {
            "GroupName": "TestGroup2",
            "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup2/UniqueID",
            "FilterExpression": "responsetime > 2"
        }
    ],
 "NextToken": "tokenstring"
}
```

Update a group with `UpdateGroup`.

```
$ aws xray update-group --group-name "TestGroup" --group-arn "arn:aws:xray:us-
east-2:123456789012:group/TestGroup/UniqueID" --filter-expression "service(\"example.com\")
 {fault OR error}"
{
    "GroupName": "TestGroup",
    "GroupARN": "arn:aws:xray:us-east-2:123456789012:group/TestGroup/UniqueID",
    "FilterExpression": "service(\"example.com\") {fault OR error}"
}
```

Delete a group with `DeleteGroup`.

```
$ aws xray delete-group --group-name "TestGroup" --group-arn "arn:aws:xray:us-
east-2:123456789012:group/TestGroup/UniqueID"
    {
    }
```

# Using sampling rules with the X-Ray API

The AWS X-Ray SDK uses the X-Ray API to get sampling rules, report sampling results, and get quotas. You can use these APIs to get a better understanding of how sampling rules work, or to implement sampling in a language that the X-Ray SDK doesn't support.

Start by getting all sampling rules with `GetSamplingRules`.

```
$ aws xray get-sampling-rules
{
    "SamplingRuleRecords": [
        {
            "SamplingRule": {
```

```
                "RuleName": "Default",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/Default",
                "ResourceARN": "*",
                "Priority": 10000,
                "FixedRate": 0.01,
                "ReservoirSize": 0,
                "ServiceName": "*",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 0.0,
            "ModifiedAt": 1530558121.0
        },
        {
            "SamplingRule": {
                "RuleName": "base-scorekeep",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/base-scorekeep",
                "ResourceARN": "*",
                "Priority": 9000,
                "FixedRate": 0.1,
                "ReservoirSize": 2,
                "ServiceName": "Scorekeep",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "*",
                "URLPath": "*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 1530573954.0,
            "ModifiedAt": 1530920505.0
        },
        {
            "SamplingRule": {
                "RuleName": "polling-scorekeep",
                "RuleARN": "arn:aws:xray:us-east-1::sampling-rule/polling-scorekeep",
                "ResourceARN": "*",
                "Priority": 5000,
                "FixedRate": 0.003,
                "ReservoirSize": 0,
                "ServiceName": "Scorekeep",
                "ServiceType": "*",
                "Host": "*",
                "HTTPMethod": "GET",
                "URLPath": "/api/state/*",
                "Version": 1,
                "Attributes": {}
            },
            "CreatedAt": 1530918163.0,
            "ModifiedAt": 1530918163.0
        }
    ]
}
```


The output includes the default rule and custom rules. See Sampling rules (p. 96) if you haven't yet created sampling rules.

Evaluate rules against incoming requests in ascending order of priority. When a rule matches, use the fixed rate and reservoir size to make a sampling decision. Record sampled requests and ignore (for tracing purposes) unsampled requests. Stop evaluating rules when a sampling decision is made.

A rules reservoir size is the target number of traces to record per second before applying the fixed rate. The reservoir applies across all services cumulatively, so you can't use it directly. However, if it is non-zero, you can borrow one trace per second from the reservoir until X-Ray assigns a quota. Before receiving a quota, record the first request each second, and apply the fixed rate to additional requests. The fixed rate is a decimal between 0 and 1.00 (100%).

The following example shows a call to `GetSamplingTargets` with details about sampling decisions made over the last 10 seconds.

```
$ aws xray get-sampling-targets --sampling-statistics-documents '[
    {
        "RuleName": "base-scorekeep",
        "ClientID": "ABCDEF1234567890ABCDEF10",
        "Timestamp": "2018-07-07T00:20:06,
        "RequestCount": 110,
        "SampledCount": 20,
        "BorrowCount": 10
    },
    {
        "RuleName": "polling-scorekeep",
        "ClientID": "ABCDEF1234567890ABCDEF10",
        "Timestamp": "2018-07-07T00:20:06",
        "RequestCount": 10500,
        "SampledCount": 31,
        "BorrowCount": 0
    }
]'
{
    "SamplingTargetDocuments": [
        {
            "RuleName": "base-scorekeep",
            "FixedRate": 0.1,
            "ReservoirQuota": 2,
            "ReservoirQuotaTTL": 1530923107.0,
            "Interval": 10
        },
        {
            "RuleName": "polling-scorekeep",
            "FixedRate": 0.003,
            "ReservoirQuota": 0,
            "ReservoirQuotaTTL": 1530923107.0,
            "Interval": 10
        }
    ],
    "LastRuleModification": 1530920505.0,
    "UnprocessedStatistics": []
}
```

The response from X-Ray includes a quota to use instead of borrowing from the reservoir. In this example, the service borrowed 10 traces from the reservoir over 10 seconds, and applied the fixed rate of 10 percent to the other 100 requests, resulting in a total of 20 sampled requests. The quota is good for five minutes (indicated by the time to live) or until a new quota is assigned. X-Ray may also assign a longer reporting interval than the default, although it didn't here.

> **Note**
> The response from X-Ray might not include a quota the first time you call it. Continue borrowing from the reservoir until you are assigned a quota.

The other two fields in the response might indicate issues with the input. Check `LastRuleModification` against the last time you called `GetSamplingRules`. If it's newer, get a new copy of the rules. `UnprocessedStatistics` can include errors that indicate that a rule has been deleted, that the statistics document in the input was too old, or permissions errors.

# AWS X-Ray segment documents

A **trace segment** is a JSON representation of a request that your application serves. A trace segment records information about the original request, information about the work that your application does locally, and **subsegments** with information about downstream calls that your application makes to AWS resources, HTTP APIs, and SQL databases.

A **segment document** conveys information about a segment to X-Ray. A segment document can be up to 64 kB and contain a whole segment with subsegments, a fragment of a segment that indicates that a request is in progress, or a single subsegment that is sent separately. You can send segment documents directly to X-Ray by using the `PutTraceSegments` API.

X-Ray compiles and processes segment documents to generate queryable **trace summaries** and **full traces** that you can access by using the `GetTraceSummaries` and `BatchGetTraces` APIs, respectively. In addition to the segments and subsegments that you send to X-Ray, the service uses information in subsegments to generate **inferred segments** and adds them to the full trace. Inferred segments represent downstream services and resources in the service map.

X-Ray provides a **JSON schema** for segment documents. You can download the schema here: xray-segmentdocument-schema-v1.0.0. The fields and objects listed in the schema are described in more detail in the following sections.

A subset of segment fields are indexed by X-Ray for use with filter expressions. For example, if you set the `user` field on a segment to a unique identifier, you can search for segments associated with specific users in the X-Ray console or by using the `GetTraceSummaries` API. For more information, see Using filter expressions to search for traces in the console (p. 59).

When you instrument your application with the X-Ray SDK, the SDK generates segment documents for you. Instead of sending segment documents directly to X-Ray, the SDK transmits them over a local UDP port to the X-Ray daemon (p. 137). For more information, see Sending segment documents to the X-Ray daemon (p. 85).

**Sections**



## Segment fields

A segment records tracing information about a request that your application serves. At a minimum, a segment records the name, ID, start time, trace ID, and end time of the request.

**Example Minimal complete segment**

```
{
  "name" : "example.com",
  "id" : "70de5b6f19ff9a0a",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
```

```
  "end_time" : 1.478293361449E9
}
```

The following fields are required, or conditionally required, for segments.

**Note**
Values must be strings (up to 250 characters) unless noted otherwise.

### Required Segment Fields

- `name` – The logical name of the service that handled the request, up to **200 characters**. For example, your application's name or domain name. Names can contain Unicode letters, numbers, and whitespace, and the following symbols: `_`, `.`, `:`, `/`, `%`, `&`, `#`, `=`, `+`, `\`, `-`, `@`
- `id` – A 64-bit identifier for the segment, unique among segments in the same trace, in **16 hexadecimal digits**.
- `trace_id` – A unique identifier that connects all segments and subsegments originating from a single client request.

  #### Trace ID Format

  A `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

  - The version number, that is, `1`.
  - The time of the original request, in Unix epoch time, in **8 hexadecimal digits**.

    For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal digits.
  - A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.

    **Trace ID Security**
    Trace IDs are visible in response headers (p. 26). Generate trace IDs with a secure random algorithm to ensure that attackers cannot calculate future trace IDs and send requests with those IDs to your application.
- `start_time` – **number** that is the time the segment was created, in floating point seconds in epoch time. For example, `1480615200.010` or `1.480615200010E9`. Use as many decimal places as you need. Microsecond resolution is recommended when available.
- `end_time` – **number** that is the time the segment was closed. For example, `1480615200.090` or `1.480615200090E9`. Specify either an `end_time` or `in_progress`.
- `in_progress` – **boolean**, set to `true` instead of specifying an `end_time` to record that a segment is started, but is not complete. Send an in-progress segment when your application receives a request that will take a long time to serve, to trace the request receipt. When the response is sent, send the complete segment to overwrite the in-progress segment. Only send one complete segment, and one or zero in-progress segments, per request.

  **Service Names**
  A segment's `name` should match the domain name or logical name of the service that generates the segment. However, this is not enforced. Any application that has permission to `PutTraceSegments` can send segments with any name.

The following fields are optional for segments.

### Optional Segment Fields

- `service` – An object with information about your application.
  - `version` – A string that identifies the version of your application that served the request.
- `user` – A string that identifies the user who sent the request.

- `origin` – The type of AWS resource running your application.

  **Supported Values**

  - `AWS::EC2::Instance` – An Amazon EC2 instance.
  - `AWS::ECS::Container` – An Amazon ECS container.
  - `AWS::ElasticBeanstalk::Environment` – An Elastic Beanstalk environment.

  When multiple values are applicable to your application, use the one that is most specific. For example, a Multicontainer Docker Elastic Beanstalk environment runs your application on an Amazon ECS container, which in turn runs on an Amazon EC2 instance. In this case you would set the origin to `AWS::ElasticBeanstalk::Environment` as the environment is the parent of the other two resources.
- `parent_id` – A subsegment ID you specify if the request originated from an instrumented application. The X-Ray SDK adds the parent subsegment ID to the tracing header (p. 26) for downstream HTTP calls. In the case of nested subsguments, a subsegment can have a segment or a subsegment as its parent.
- `http` – `http` (p. 108) objects with information about the original HTTP request.
- `aws` – `aws` (p. 111) object with information about the AWS resource on which your application served the request.
- `error`, `throttle`, `fault`, and `cause` – error (p. 113) fields that indicate an error occurred and that include information about the exception that caused the error.
- `annotations` – `annotations` (p. 110) object with key-value pairs that you want X-Ray to index for search.
- `metadata` – `metadata` (p. 110) object with any additional data that you want to store in the segment.
- `subsegments` – **array** of `subsegment` (p. 105) objects.

# Subsegments

You can create subsegments to record calls to AWS services and resources that you make with the AWS SDK, calls to internal or external HTTP web APIs, or SQL database queries. You can also create subsegments to debug or annotate blocks of code in your application. Subsegments can contain other subsegments, so a custom subsegment that records metadata about an internal function call can contain other custom subsegments and subsegments for downstream calls.

A subsegment records a downstream call from the point of view of the service that calls it. X-Ray uses subsegments to identify downstream services that don't send segments and create entries for them on the service graph.

A subsegment can be embedded in a full segment document or sent independently. Send subsegments separately to asynchronously trace downstream calls for long-running requests, or to avoid exceeding the maximum segment document size.

### Example Segment with embedded subsegment

An independent subsegment has a `type` of `subsegment` and a `parent_id` that identifies the parent segment.

```
{
  "trace_id"   : "1-5759e988-bd862e3fe1be46a994272793",
  "id"         : "defdfd9912dc5a56",
  "start_time" : 1461096053.37518,
  "end_time"   : 1461096053.4042,
  "name"       : "www.example.com",
```

```
  "http"       : {
    "request"  : {
      "url"        : "https://www.example.com/health",
      "method"     : "GET",
      "user_agent" : "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6) AppleWebKit/601.7.7",
      "client_ip"  : "11.0.3.111"
    },
    "response" : {
      "status"         : 200,
      "content_length" : 86
    }
  },
  "subsegments" : [
    {
      "id"         : "53995c3f42cd8ad8",
      "name"       : "api.example.com",
      "start_time" : 1461096053.37769,
      "end_time"   : 1461096053.40379,
      "namespace"  : "remote",
      "http"       : {
        "request"  : {
          "url"    : "https://api.example.com/health",
          "method" : "POST",
          "traced" : true
        },
        "response" : {
          "status"         : 200,
          "content_length" : 861
        }
      }
    }
  ]
}
```

For long-running requests, you can send an in-progress segment to notify X-Ray that the request was received, and then send subsegments separately to trace them before completing the original request.

### Example In-progress segment

```
{
  "name" : "example.com",
  "id" : "70de5b6f19ff9a0b",
  "start_time" : 1.478293361271E9,
  "trace_id" : "1-581cf771-a006649127e371903a2de979",
  "in_progress": true
}
```

### Example Independent subsegment

An independent subsegment has a `type` of `subsegment`, a `trace_id`, and a `parent_id` that identifies the parent segment.

```
{
  "name" : "api.example.com",
  "id" : "53995c3f42cd8ad8",
  "start_time" : 1.478293361271E9,
  "end_time" : 1.478293361449E9,
  "type" : "subsegment",
  "trace_id" : "1-581cf771-a006649127e371903a2de979"
  "parent_id" : "defdfd9912dc5a56",
  "namespace"  : "remote",
  "http"       : {
      "request"  : {
```

```
            "url"    : "https://api.example.com/health",
            "method" : "POST",
            "traced" : true
        },
        "response" : {
            "status"         : 200,
            "content_length" : 861
        }
    }
}
```

When the request is complete, close the segment by resending it with an `end_time`. The complete segment overwrites the in-progress segment.

You can also send subsegments separately for completed requests that triggered asynchronous workflows. For example, a web API may return a `OK 200` response immediately prior to starting the work that the user requested. You can send a full segment to X-Ray as soon as the response is sent, followed by subsegments for work completed later. As with segments, you can also send a subsegment fragment to record that the subsegment has started, and then overwrite it with a full subsegment once the downstream call is complete.

The following fields are required, or are conditionally required, for subsegments.

**Note**
Values are strings up to 250 characters unless noted otherwise.

### Required Subsegment Fields

- `id` – A 64-bit identifier for the subsegment, unique among segments in the same trace, in **16 hexadecimal digits**.
- `name` – The logical name of the subsegment. For downstream calls, name the subsegment after the resource or service called. For custom subsegments, name the subsegment after the code that it instruments (e.g., a function name).
- `start_time` – **number** that is the time the subsegment was created, in floating point seconds in epoch time, accurate to milliseconds. For example, `1480615200.010` or `1.480615200010E9`.
- `end_time` – **number** that is the time the subsegment was closed. For example, `1480615200.090` or `1.480615200090E9`. Specify an `end_time` or `in_progress`.
- `in_progress` – **boolean** that is set to `true` instead of specifying an `end_time` to record that a subsegment is started, but is not complete. Only send one complete subsegment, and one or zero in-progress subsegments, per downstream request.
- `trace_id` – Trace ID of the subsegment's parent segment. Required only if sending a subsegment separately.

  #### Trace ID Format

  A `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

  - The version number, that is, `1`.
  - The time of the original request, in Unix epoch time, in **8 hexadecimal digits**.

    For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal digits.
  - A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.
- `parent_id` – Segment ID of the subsegment's parent segment. Required only if sending a subsegment separately. In the case of nested subsegments, a subsegment can have a segment or a subsegment as its parent.
- `type` – `subsegment`. Required only if sending a subsegment separately.

The following fields are optional for subsegments.

**Optional Subsegment Fields**

- `namespace` – `aws` for AWS SDK calls; `remote` for other downstream calls.
- `http` – `http` (p. 108) object with information about an outgoing HTTP call.
- `aws` – `aws` (p. 111) object with information about the downstream AWS resource that your application called.
- `error`, `throttle`, `fault`, and `cause` – error (p. 113) fields that indicate an error occurred and that include information about the exception that caused the error.
- `annotations` – `annotations` (p. 110) object with key-value pairs that you want X-Ray to index for search.
- `metadata` – `metadata` (p. 110) object with any additional data that you want to store in the segment.
- `subsegments` – **array** of `subsegment` (p. 105) objects.
- `precursor_ids` – **array** of subsegment IDs that identifies subsegments with the same parent that completed prior to this subsegment.

# HTTP request data

Use an HTTP block to record details about an HTTP request that your application served (in a segment) or that your application made to a downstream HTTP API (in a subsegment). Most of the fields in this object map to information found in an HTTP request and response.

**`http`**

All fields are optional.

- `request` – Information about a request.
  - `method` – The request method. For example, `GET`.
  - `url` – The full URL of the request, compiled from the protocol, hostname, and path of the request.
  - `user_agent` – The user agent string from the requester's client.
  - `client_ip` – The IP address of the requester. Can be retrieved from the IP packet's `Source Address` or, for forwarded requests, from an `X-Forwarded-For` header.
  - `x_forwarded_for` – (segments only) **boolean** indicating that the `client_ip` was read from an `X-Forwarded-For` header and is not reliable as it could have been forged.
  - `traced` – (subsegments only) **boolean** indicating that the downstream call is to another traced service. If this field is set to `true`, X-Ray considers the trace to be broken until the downstream service uploads a segment with a `parent_id` that matches the `id` of the subsegment that contains this block.
- `response` – Information about a response.
  - `status` – **number** indicating the HTTP status of the response.
  - `content_length` – **number** indicating the length of the response body in bytes.

When you instrument a call to a downstream web api, record a subsegment with information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

**Example Segment for HTTP call served by an application running on Amazon EC2**

```
{
```

```
  "id": "6b55dcc497934f1a",
  "start_time": 1484789387.126,
  "end_time": 1484789387.535,
  "trace_id": "1-5880168b-fd5158284b67678a3bb5a78c",
  "name": "www.example.com",
  "origin": "AWS::EC2::Instance",
  "aws": {
    "ec2": {
      "availability_zone": "us-west-2c",
      "instance_id": "i-0b5a4678fc325bg98"
    },
    "xray": {
        "sdk_version": "2.4.0 for Java"
    },
  },
  "http": {
    "request": {
      "method": "POST",
      "client_ip": "78.255.233.48",
      "url": "http://www.example.com/api/user",
      "user_agent": "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101
 Firefox/45.0",
      "x_forwarded_for": true
    },
    "response": {
      "status": 200
    }
  }
```

**Example Subsegment for a downstream HTTP call**

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

**Example Inferred segment for a downstream HTTP call**

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-5880168b-fd5153bb58284b67678aa78c",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
```

```
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

# Annotations

Segments and subsegments can include an `annotations` object containing one or more fields that X-Ray indexes for use with filter expressions. Fields can have string, number, or Boolean values (no objects or arrays). X-Ray indexes up to 50 annotations per trace.

**Example Segment for HTTP call with annotations**

```
{
  "id": "6b55dcc497932f1a",
  "start_time": 1484789187.126,
  "end_time": 1484789187.535,
  "trace_id": "1-5880168b-fd515828bs07678a3bb5a78c",
  "name": "www.example.com",
  "origin": "AWS::EC2::Instance",
  "aws": {
    "ec2": {
      "availability_zone": "us-west-2c",
      "instance_id": "i-0b5a4678fc325bg98"
    },
    "xray": {
        "sdk_version": "2.4.0 for Java"
    },
  },
  "annotations": {
    "customer_category" : 124,
    "zip_code" : 98101,
    "country" : "United States",
    "internal" : false
  },
  "http": {
    "request": {
      "method": "POST",
      "client_ip": "78.255.233.48",
      "url": "http://www.example.com/api/user",
      "user_agent": "Mozilla/5.0 (Windows NT 6.1; WOW64; rv:45.0) Gecko/20100101
 Firefox/45.0",
      "x_forwarded_for": true
    },
    "response": {
      "status": 200
    }
  }
```

Keys must be alphanumeric in order to work with filters. Underscore is allowed. Other symbols and whitespace are not allowed.

# Metadata

Segments and subsegments can include a `metadata` object containing one or more fields with values of any type, including objects and arrays. X-Ray does not index metadata, and values can be any size, as long as the segment document doesn't exceed the maximum size (64 kB). You can view metadata in

the full segment document returned by the `BatchGetTraces` API. Field keys (`debug` in the following example) starting with `AWS.` are reserved for use by AWS-provided SDKs and clients.

**Example Custom subsegment with metadata**

```
{
  "id": "0e58d2918e9038e8",
  "start_time": 1484789387.502,
  "end_time": 1484789387.534,
  "name": "## UserModel.saveUser",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
  "subsegments": [
    {
      "id": "0f910026178b71eb",
      "start_time": 1484789387.502,
      "end_time": 1484789387.534,
      "name": "DynamoDB",
      "namespace": "aws",
      "http": {
        "response": {
          "content_length": 58,
          "status": 200
        }
      },
      "aws": {
        "table_name": "scorekeep-user",
        "operation": "UpdateItem",
        "request_id": "3AIENM5J4ELQ3SPODHKBIRVIC3VV4KQNSO5AEMVJF66Q9ASUAAJG",
        "resource_names": [
          "scorekeep-user"
        ]
      }
    }
  ]
}
```

# AWS resource data

For segments, the `aws` object contains information about the resource on which your application is running. Multiple fields can apply to a single resource. For example, an application running in a multicontainer Docker environment on Elastic Beanstalk could have information about the Amazon EC2 instance, the Amazon ECS container running on the instance, and the Elastic Beanstalk environment itself.

### `aws` (Segments)

All fields are optional.

- `account_id` – If your application sends segments to a different AWS account, record the ID of the account running your application.
- `ecs` – Information about an Amazon ECS container.
  - `container` – The container ID of the container running your application.
- `ec2` – Information about an EC2 instance.
  - `instance_id` – The instance ID of the EC2 instance.
  - `availability_zone` – The Availability Zone in which the instance is running.

### Example AWS block with plugins

```
"aws": {
  "elastic_beanstalk": {
    "version_label": "app-5a56-170119_190650-stage-170119_190650",
    "deployment_id": 32,
    "environment_name": "scorekeep"
  },
  "ec2": {
    "availability_zone": "us-west-2c",
    "instance_id": "i-075ad396f12bc325a"
  },
  "xray": {
    "sdk": "2.4.0 for Java"
  }
}
```

- `elastic_beanstalk` – Information about an Elastic Beanstalk environment. You can find this information in a file named `/var/elasticbeanstalk/xray/environment.conf` on the latest Elastic Beanstalk platforms.
  - `environment_name` – The name of the environment.
  - `version_label` – The name of the application version that is currently deployed to the instance that served the request.
  - `deployment_id` – **number** indicating the ID of the last successful deployment to the instance that served the request.

For subsegments, record information about the AWS services and resources that your application accesses. X-Ray uses this information to create inferred segments that represent the downstream services in your service map.

#### `aws` (Subsegments)

All fields are optional.

- `operation` – The name of the API action invoked against an AWS service or resource.
- `account_id` – If your application accesses resources in a different account, or sends segments to a different account, record the ID of the account that owns the AWS resource that your application accessed.
- `region` – If the resource is in a region different from your application, record the region. For example, `us-west-2`.
- `request_id` – Unique identifier for the request.
- `queue_url` – For operations on an Amazon SQS queue, the queue's URL.
- `table_name` – For operations on a DynamoDB table, the name of the table.

### Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
```

```
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

# Errors and exceptions

When an error occurs, you can record details about the error and exceptions that it generated. Record errors in segments when your application returns an error to the user, and in subsegments when a downstream call returns an error.

### error types

Set one or more of the following fields to `true` to indicate that an error occurred. Multiple types can apply if errors compound. For example, a `429 Too Many Requests` error from a downstream call may cause your application to return `500 Internal Server Error`, in which case all three types would apply.

- `error` – **boolean** indicating that a client error occurred (response status code was 4XX Client Error).
- `throttle` – **boolean** indicating that a request was throttled (response status code was *429 Too Many Requests*).
- `fault` – **boolean** indicating that a server error occurred (response status code was 5XX Server Error).

Indicate the cause of the error by including a **cause** object in the segment or subsegment.

#### `cause`

A cause can be either a **16 character** exception ID or an object with the following fields:

- `working_directory` – The full path of the working directory when the exception occurred.
- `paths` – The **array** of paths to libraries or modules in use when the exception occurred.
- `exceptions` – The **array** of **exception** objects.

Include detailed information about the error in one or more **exception** objects.

#### `exception`

All fields are optional except `id`.

- `id` – A 64-bit identifier for the exception, unique among segments in the same trace, in **16 hexadecimal digits**.
- `message` – The exception message.
- `type` – The exception type.
- `remote` – **boolean** indicating that the exception was caused by an error returned by a downstream service.
- `truncated` – **integer** indicating the number of stack frames that are omitted from the `stack`.
- `skipped` – **integer** indicating the number of exceptions that were skipped between this exception and its child, that is, the exception that it caused.
- `cause` – Exception ID of the exception's parent, that is, the exception that caused this exception.
- `stack` – **array** of **stackFrame** objects.

If available, record information about the call stack in **stackFrame** objects.

**`stackFrame`**

All fields are optional.

- `path` – The relative path to the file.
- `line` – The line in the file.
- `label` – The function or method name.

# SQL queries

You can create subsegments for queries that your application makes to an SQL database.

**`sql`**

All fields are optional.

- `connection_string` – For SQL Server or other database connections that don't use URL connection strings, record the connection string, excluding passwords.
- `url` – For a database connection that uses a URL connection string, record the URL, excluding passwords.
- `sanitized_query` – The database query, with any user provided values removed or replaced by a placeholder.
- `database_type` – The name of the database engine.
- `database_version` – The version number of the database engine.
- `driver_version` – The name and version number of the database engine driver that your application uses.
- `user` – The database username.
- `preparation` – `call` if the query used a `PreparedCall`; `statement` if the query used a `PreparedStatement`.

### Example Subsegment with an SQL Query

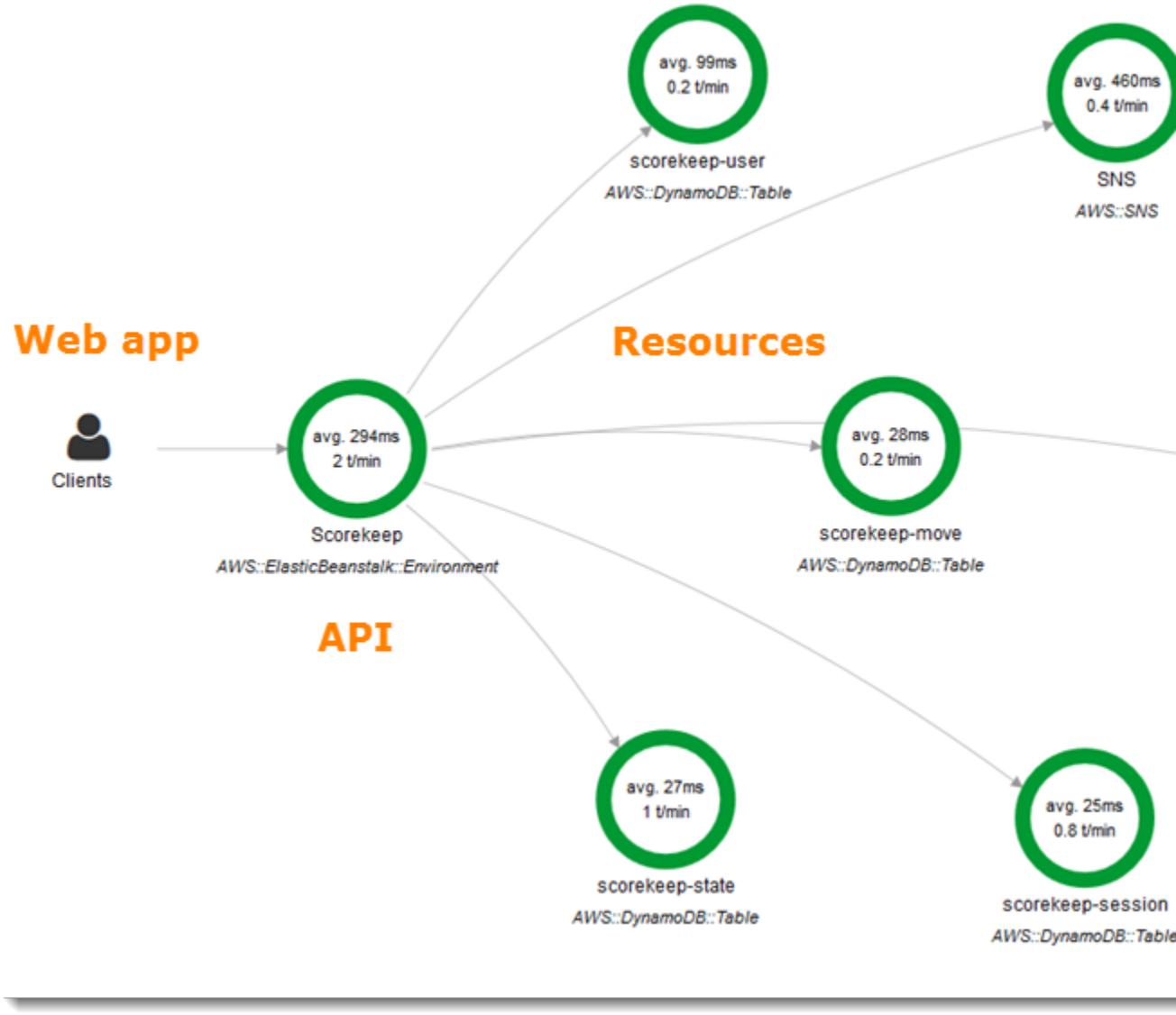
```
{
  "id": "3fd8634e78ca9560",
  "start_time": 1484872218.696,
  "end_time": 1484872218.697,
  "name": "ebdb@aawijb5u25wdoy.cpamxznpdoq8.us-west-2.rds.amazonaws.com",
  "namespace": "remote",
  "sql" : {
    "url": "jdbc:postgresql://aawijb5u25wdoy.cpamxznpdoq8.us-west-2.rds.amazonaws.com:5432/
ebdb",
    "preparation": "statement",
    "database_type": "PostgreSQL",
    "database_version": "9.5.4",
    "driver_version": "PostgreSQL 9.4.1211.jre7",
    "user" : "dbuser",
    "sanitized_query" : "SELECT  *  FROM  customers  WHERE  customer_id=?;"
  }
}
```

# AWS X-Ray sample application

The AWS X-Ray eb-java-scorekeep sample app, available on GitHub, shows the use of the AWS X-Ray SDK to instrument incoming HTTP calls, DynamoDB SDK clients, and HTTP clients. The sample app uses AWS Elastic Beanstalk features to create DynamoDB tables, compile Java code on instance, and run the X-Ray daemon without any additional configuration.


The sample is an instrumented version of the Scorekeep project on AWSLabs. It includes a front-end web app, the API that it calls, and the DynamoDB tables that it uses to store data. All the components are hosted in an Elastic Beanstalk environment for portability and ease of deployment.

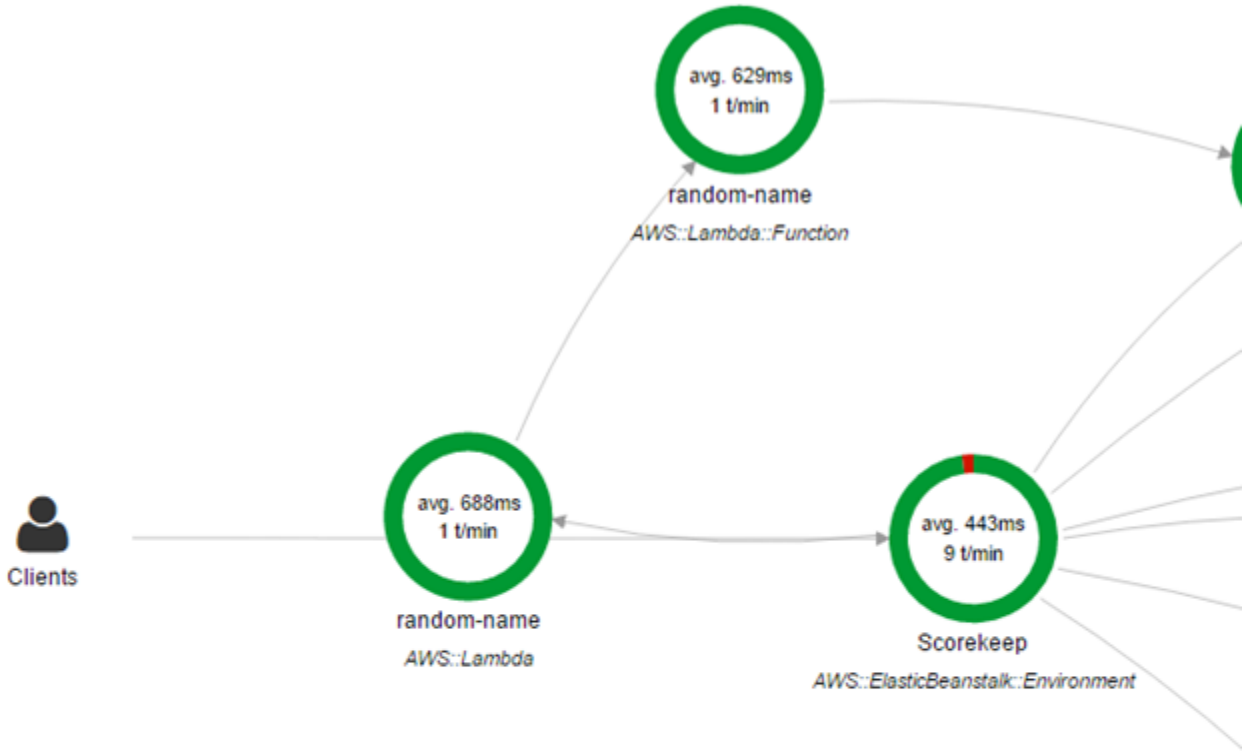
Basic instrumentation with filters (p. 196), plugins (p. 189), and instrumented AWS SDK clients (p. 199) is shown in the project's `xray-gettingstarted` branch. This is the branch that you deploy in the getting started tutorial (p. 8). Because this branch only includes the basics, you can diff it against the `master` branch to quickly understand the basics.




The sample application shows basic instrumentation in these files:

- **HTTP request filter** – `WebConfig.java`
- **AWS SDK client instrumentation** – `build.gradle`

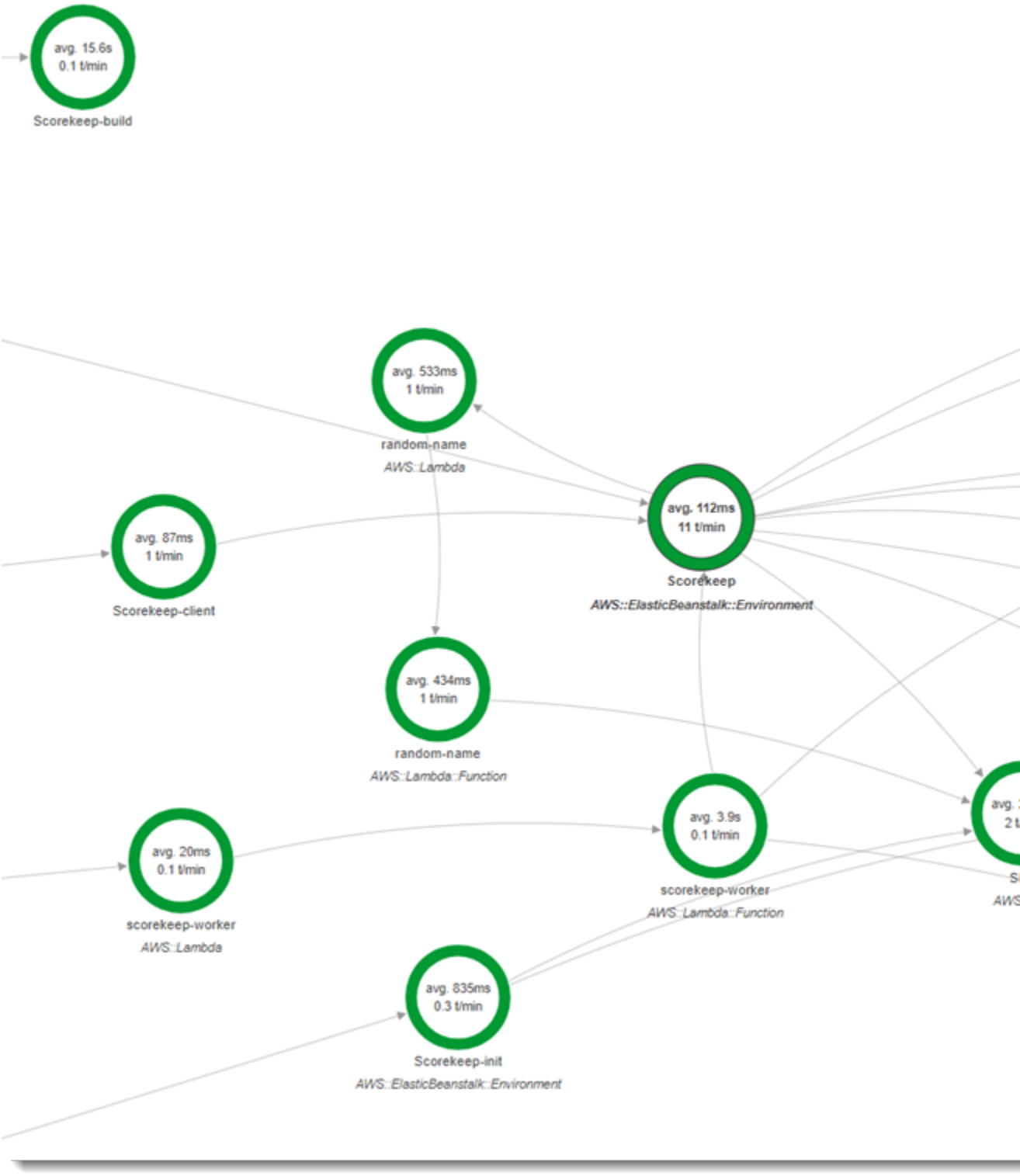
The `xray` branch of the application adds the use of HTTPClient (p. 200), Annotations (p. 205), SQL queries (p. 202), custom subsegments (p. 203), an instrumented AWS Lambda (p. 167) function, and instrumented initialization code and scripts (p. 128). This service map shows the `xray` branch running without a connected SQL database:




To support user log-in and AWS SDK for JavaScript use in the browser, the `xray-cognito` branch adds Amazon Cognito to support user authentication and authorization. With credentials retrieved from Amazon Cognito, the web app also sends trace data to X-Ray to record request information from the client's point of view. The browser client appears as its own node on the service map, and records additional information, including the URL of the page that the user is viewing, and the user's ID.

Finally, the `xray-worker` branch adds an instrumented Python Lambda function that runs independently, processing items from an Amazon SQS queue. Scorekeep adds an item to the queue each time a game ends. The Lambda worker, triggered by CloudWatch Events, pulls items from the queue every few minutes and processes them to store game records in Amazon S3 for analysis.

With all features enabled, Scorekeep's service map looks like this:




For instructions on using the sample application with X-Ray, see the getting started tutorial (p. 8). In addition to the basic use of the X-Ray SDK for Java discussed in the tutorial, the sample also shows how to use the following features.

**Advanced Features**



# Manually instrumenting AWS SDK clients

The X-Ray SDK for Java automatically instruments all AWS SDK clients when you include the AWS SDK Instrumentor submodule in your build dependencies (p. 188).

You can disable automatic client instrumentation by removing the Instrumentor submodule. This enables you to instrument some clients manually while ignoring others, or use different tracing handlers on different clients.

To illustrate support for instrumenting specific AWS SDK clients, the application passes a tracing handler to `AmazonDynamoDBClientBuilder` as a request handler in the user, game, and session model. This code change tells the SDK to instrument all calls to DynamoDB using those clients.

**Example `src/main/java/scorekeep/SessionModel.java` – Manual AWS SDK client instrumentation**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.handlers.TracingHandler;

public class SessionModel {
  private AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
        .withRegion(Constants.REGION)
        .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
        .build();
  private DynamoDBMapper mapper = new DynamoDBMapper(client);
```

If you remove the AWS SDK Instrumentor submodule from project dependencies, only the manually instrumented AWS SDK clients appear in the service map.

# Creating additional subsegments

In the user model class, the application manually creates subsegments to group all downstream calls made within the `saveUser` function and adds metadata.

**Example `src/main/java/scorekeep/UserModel.java` - Custom subsegments**

```
import com.amazonaws.xray.AWSXRay;
```

```
import com.amazonaws.xray.entities.Subsegment;
...
    public void saveUser(User user) {
    // Wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## UserModel.saveUser");
    try {
      mapper.save(user);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

# Recording annotations, metadata, and user IDs

In the game model class, the application records `Game` objects in a metadata (p. 206) block each time it saves a game in DynamoDB. Separately, the application records game IDs in annotations (p. 205) for use with filter expressions (p. 59).

**Example `src/main/java/scorekeep/GameModel.java` – Annotations and metadata**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

In the move controller, the application records user IDs (p. 207) with `setUser`. User IDs are recorded in a separate field on segments and are indexed for use with search.

**Example src/main/java/scorekeep/MoveController.java – User ID**

```
import com.amazonaws.xray.AWSXRay;
...
  @RequestMapping(value="/{userId}", method=RequestMethod.POST)
  public Move newMove(@PathVariable String sessionId, @PathVariable String gameId,
 @PathVariable String userId, @RequestBody String move) throws SessionNotFoundException,
 GameNotFoundException, StateNotFoundException, RulesException {
    AWSXRay.getCurrentSegment().setUser(userId);
```

```
    return moveFactory.newMove(sessionId, gameId, userId, move);
  }
```

# Instrumenting outgoing HTTP calls

The user factory class shows how the application uses the X-Ray SDK for Java's version of `HTTPClientBuilder` to instrument outgoing HTTP calls.

**Example `src/main/java/scorekeep/UserFactory.java` – HTTPClient instrumentation**

```
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;

  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
    HttpGet httpGet = new HttpGet("http://uinames.com/api/");
    CloseableHttpResponse response = httpclient.execute(httpGet);
    try {
      HttpEntity entity = response.getEntity();
      InputStream inputStream = entity.getContent();
      ObjectMapper mapper = new ObjectMapper();
      Map<String, String> jsonMap = mapper.readValue(inputStream, Map.class);
      String name = jsonMap.get("name");
      EntityUtils.consume(entity);
      return name;
    } finally {
      response.close();
    }
  }
```

If you currently use `org.apache.http.impl.client.HttpClientBuilder`, you can simply swap out the import statement for that class with one for `com.amazonaws.xray.proxies.apache.http.HttpClientBuilder`.

# Instrumenting calls to a PostgreSQL database

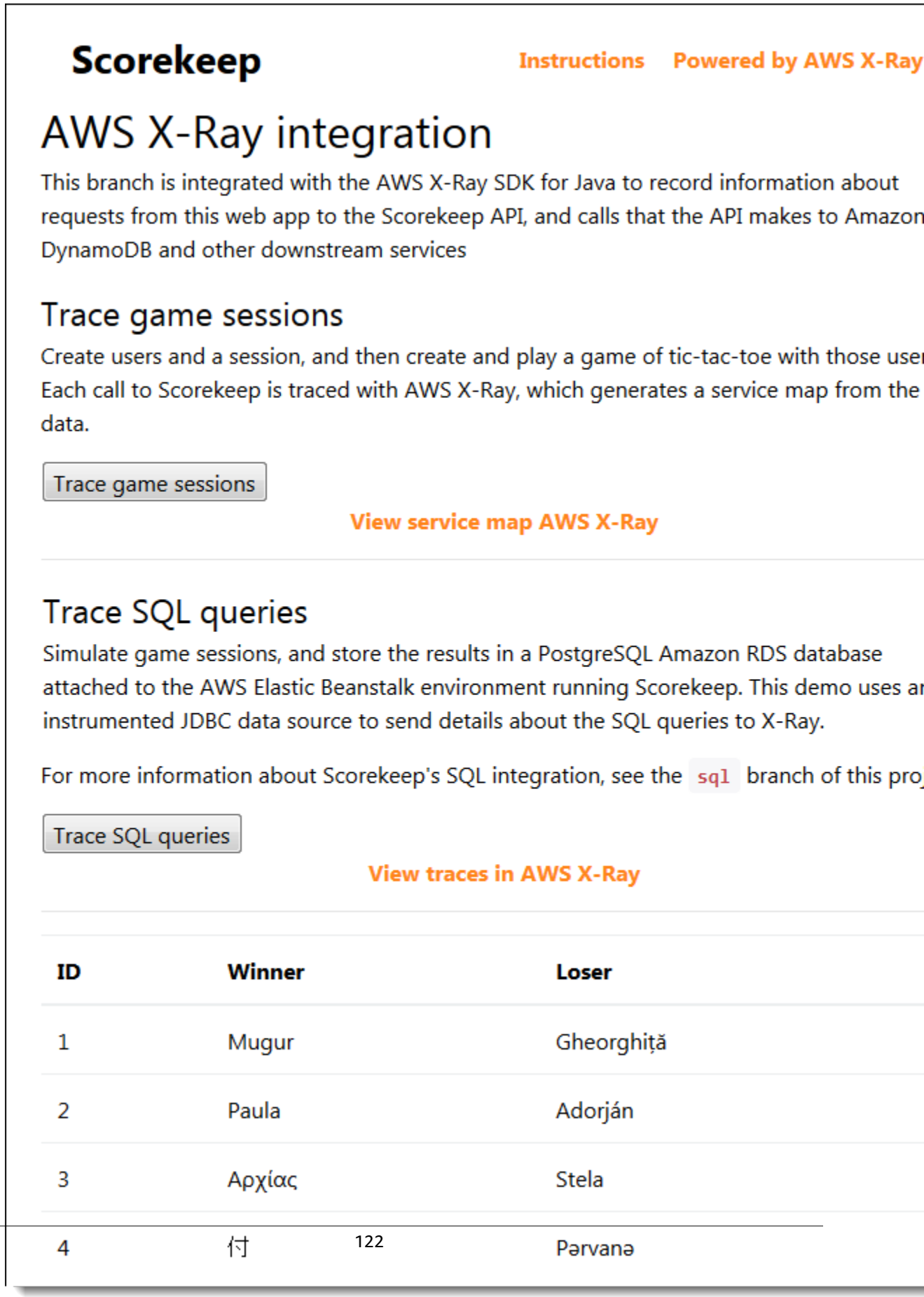
The `application-pgsql.properties` file adds the X-Ray PostgreSQL tracing interceptor to the data source created in `RdsWebConfig.java`.

**Example `application-pgsql.properties` – PostgreSQL database instrumentation**

```
spring.datasource.continue-on-error=true
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.jdbc-interceptors=com.amazonaws.xray.sql.postgres.TracingInterceptor
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL94Dialect
```

> **Note**
> See Configuring Databases with Elastic Beanstalk in the *AWS Elastic Beanstalk Developer Guide* for details on how to add a PostgreSQL database to the application environment.

The X-Ray demo page in the `xray` branch includes a demo that uses the instrumented data source to generate traces that show information about the SQL queries that it generates. Navigate to the `/#/xray` path in the running application or choose **Powered by AWS X-Ray** in the navigation bar to see the demo page.



Scorekeep

Instructions Powered by AWS X-Ray

# AWS X-Ray integration

This branch is integrated with the AWS X-Ray SDK for Java to record information about requests from this web app to the Scorekeep API, and calls that the API makes to Amazon DynamoDB and other downstream services

## Trace game sessions

Create users and a session, and then create and play a game of tic-tac-toe with those user Each call to Scorekeep is traced with AWS X-Ray, which generates a service map from the data.

Trace game sessions

View service map AWS X-Ray

## Trace SQL queries

Simulate game sessions, and store the results in a PostgreSQL Amazon RDS database attached to the AWS Elastic Beanstalk environment running Scorekeep. This demo uses ar instrumented JDBC data source to send details about the SQL queries to X-Ray.

For more information about Scorekeep's SQL integration, see the `sql` branch of this proj

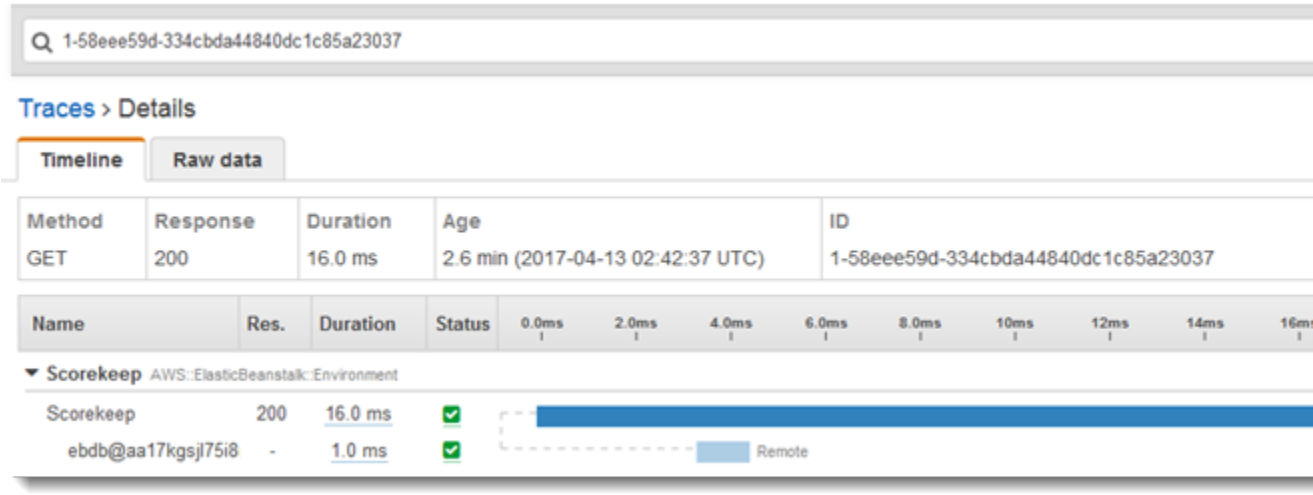
Trace SQL queries

View traces in AWS X-Ray

| ID | Winner | Loser |
|---|---|---|
| 1 | Mugur | Gheorghiță |
| 2 | Paula | Adorján |
| 3 | Αρχίας | Stela |
| 4 | 付 | Pərvanə |

Choose **Trace SQL queries** to simulate game sessions and store the results in the attached database. Then, choose **View traces in AWS X-Ray** to see a filtered list of traces that hit the API's `/api/history` route.

Choose one of the traces from the list to see the timeline, including the SQL query.




# Instrumenting AWS Lambda functions

Scorekeep uses two AWS Lambda functions. The first is a Node.js function from the `lambda` branch that generates random names for new users. When a user creates a session without entering a name, the application calls a function named `random-name` with the AWS SDK for Java. The X-Ray SDK for Java records information about the call to Lambda in a subsegment like any other call made with an instrumented AWS SDK client.

> **Note**
> Running the `random-name` Lambda function requires the creation of additional resources outside of the Elastic Beanstalk environment. See the readme for more information and instructions: AWS Lambda Integration.

The second function, `scorekeep-worker`, is a Python function that runs independently of the Scorekeep API. When a game ends, the API writes the session ID and game ID to an SQS queue. The worker function reads items from the queue, and calls the Scorekeep API to construct complete records of each game session for storage in Amazon S3.

Scorekeep includes AWS CloudFormation templates and scripts to create both functions. Because you need to bundle the X-Ray SDK with the function code, the templates create the functions without any code. When you deploy Scorekeep, a configuration file included in the `.ebextensions` folder creates a source bundle that includes the SDK, and updates the function code and configuration with the AWS Command Line Interface.

**Functions**

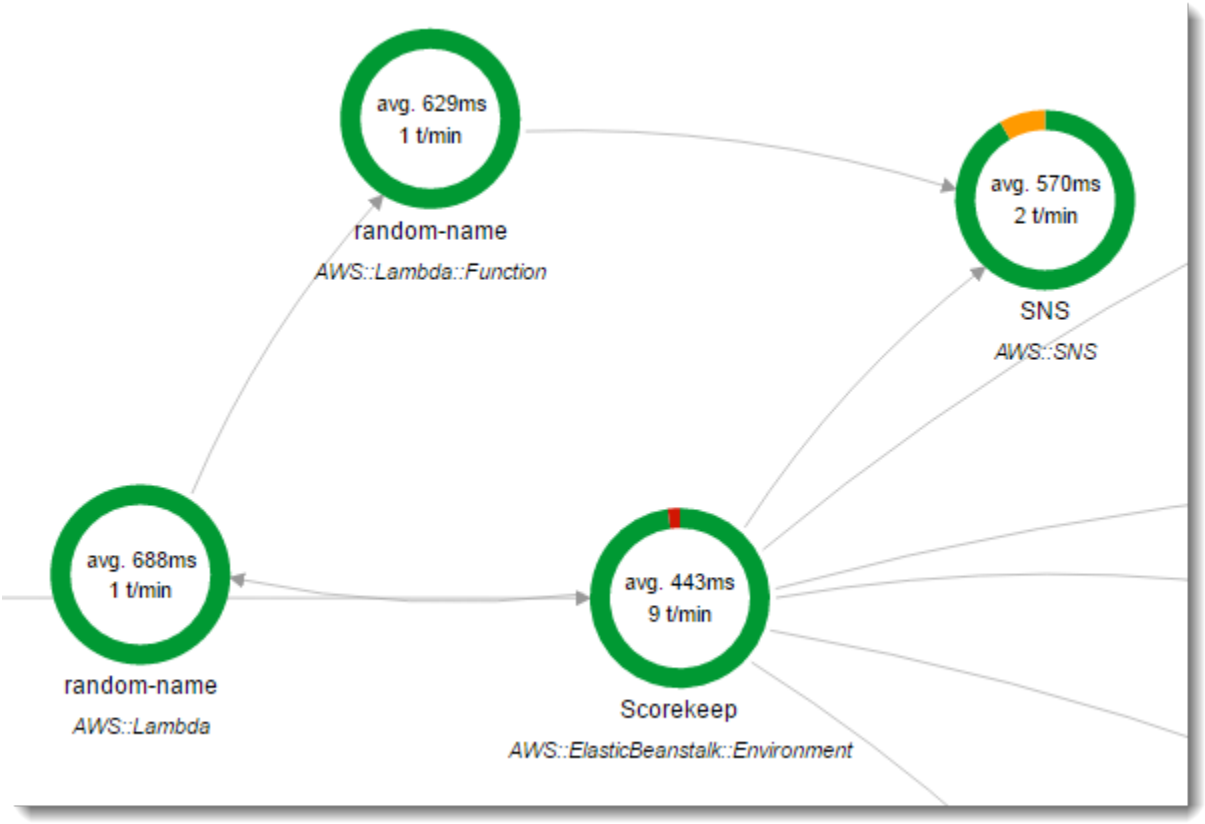
- Random name (p. 124)
- Worker (p. 125)

# Random name

Scorekeep calls the random name function when a user starts a game session without signing in or specifying a user name. When Lambda processes the call to `random-name`, it reads the tracing header (p. 26), which contains the trace ID and sampling decision written by the X-Ray SDK for Java.

For each sampled request, Lambda runs the X-Ray daemon and writes two segments. The first segment records information about the call to Lambda that invokes the function. This segment contains the same information as the subsegment recorded by Scorekeep, but from the Lambda point of view. The second segment represents the work that the function does.

Lambda passes the function segment to the X-Ray SDK through the function context. When you instrument a Lambda function, you don't use the SDK to create a segment for incoming requests (p. 219). Lambda provides the segment, and you use the SDK to instrument clients and write subsegments.




The `random-name` function is implemented in Node.js. It uses the SDK for JavaScript in Node.js to send notifications with Amazon SNS, and the X-Ray SDK for Node.js to instrument the AWS SDK client. To write annotations, the function creates a custom subsegment with `AWSXRay.captureFunc`, and writes annotations in the instrumented function. In Lambda, you can't write annotations directly to the function segment, only to a subsegment that you create.

**Example `function/index.js` -- Random name Lambda function**

```
var AWSXRay = require('aws-xray-sdk-core');
var AWS = AWSXRay.captureAWS(require('aws-sdk'));

AWS.config.update({region: process.env.AWS_REGION});
```

```
var Chance = require('chance');

var myFunction = function(event, context, callback) {
  var sns = new AWS.SNS();
  var chance = new Chance();
  var userid = event.userid;
  var name = chance.first();

  AWSXRay.captureFunc('annotations', function(subsegment){
    subsegment.addAnnotation('Name', name);
    subsegment.addAnnotation('UserID', event.userid);
  });

  // Notify
  var params = {
    Message: 'Created randon name "' + name + '"" for user "' + userid + '".',
    Subject: 'New user: ' + name,
    TopicArn: process.env.TOPIC_ARN
  };
  sns.publish(params, function(err, data) {
    if (err) {
      console.log(err, err.stack);
      callback(err);
    }
    else {
      console.log(data);
      callback(null, {"name": name});
    }
  });
};

exports.handler = myFunction;
```

This function is created automatically when you deploy the sample application to Elastic Beanstalk. The `xray` branch includes a script to create a blank Lambda function. Configuration files in the `.ebextensions` folder build the function package with `npm install` during deployment, and then update the Lambda function with the AWS CLI.

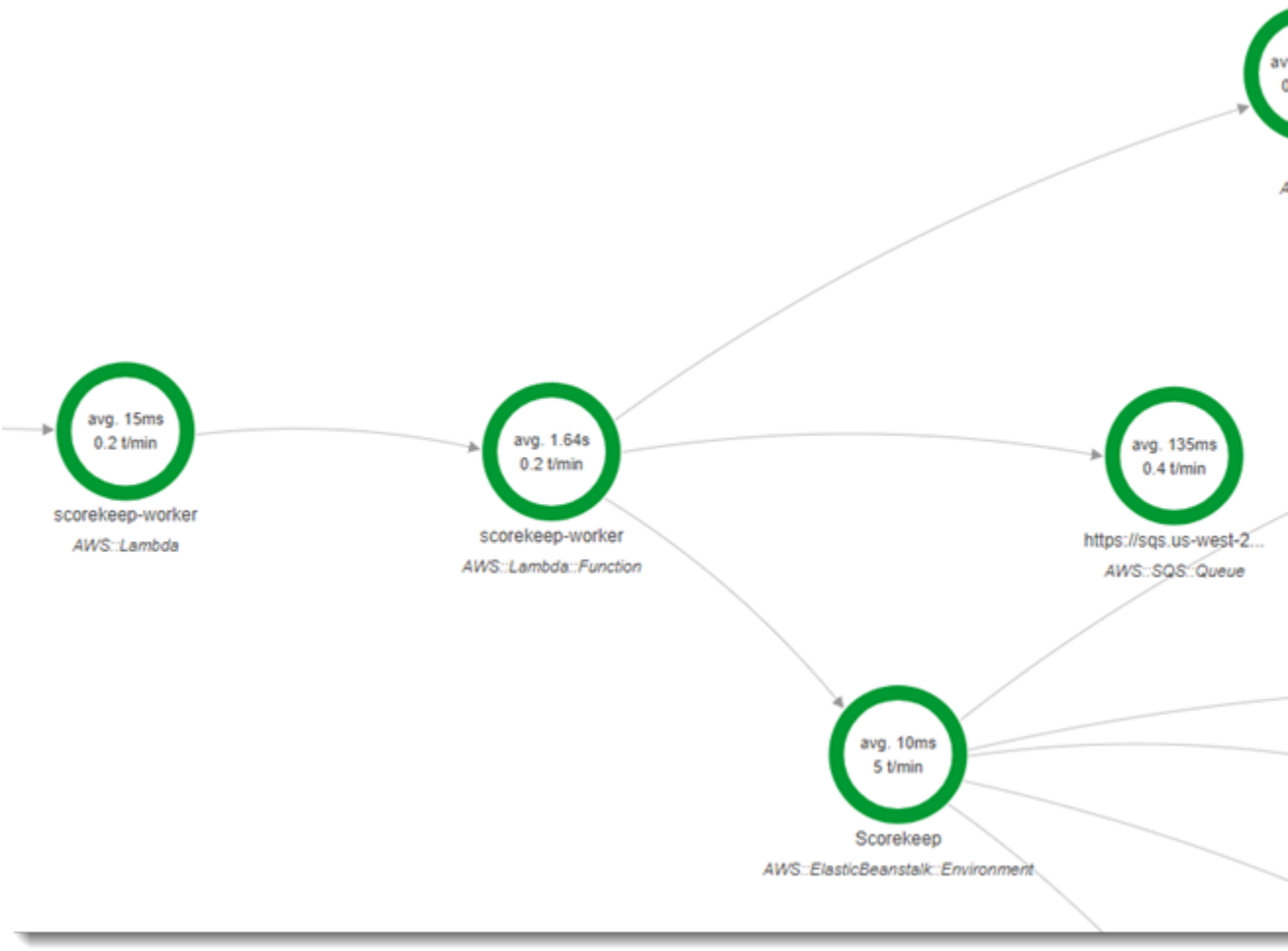
# Worker

The instrumented worker function is provided in its own branch, `xray-worker`, as it cannot run unless you create the worker function and related resources first. See the branch readme for instructions.

The function is triggered by a bundled Amazon CloudWatch Events event every 5 minutes. When it runs, the function pulls an item from an Amazon SQS queue that Scorekeep manages. Each message contains information about a completed game.

The worker pulls the game record and documents from other tables that the game record references. For example, the game record in DynamoDB includes a list of moves that were executed during the game. The list does not contain the moves themselves, but rather IDs of moves that are stored in a separate table.

Sessions, and states are stored as references as well. This keeps the entries in the game table from being too large, but requires additional calls to get all of the information about the game. The worker dereferences all of these entries and constructs a complete record of the game as a single document in Amazon S3. When you want to do analytics on the data, you can run queries on it directly in Amazon S3 with Amazon Athena without running read-heavy data migrations to get your data out of DynamoDB.




The worker function has active tracing enabled in its configuration in AWS Lambda. Unlike the random name function, the worker does not receive a request from an instrumented application, so AWS Lambda doesn't receive a tracing header. With active tracing, Lambda creates the trace ID and makes sampling decisions.

The X-Ray SDK for Python is just a few lines at the top of the function that import the SDK and run its `patch_all` function to patch the AWS SDK for Python (Boto) and HTTclients that it uses to call Amazon SQS and Amazon S3. When the worker calls the Scorekeep API, the SDK adds the tracing header (p. 26) to the request to trace calls through the API.

**Example `_lambda/scorekeep-worker/scorekeep-worker.py` -- Worker Lambda function**

```
import os
import boto3
import json
import requests
import time
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

patch_all()
queue_url = os.environ['WORKER_QUEUE']
```

```
def lambda_handler(event, context):
    # Create SQS client
    sqs = boto3.client('sqs')
    s3client = boto3.client('s3')

    # Receive message from SQS queue
    response = sqs.receive_message(
        QueueUrl=queue_url,
        AttributeNames=[
            'SentTimestamp'
        ],
        MaxNumberOfMessages=1,
        MessageAttributeNames=[
            'All'
        ],
        VisibilityTimeout=0,
        WaitTimeSeconds=0
    )
   ...
```

# Instrumenting Amazon ECS applications

In the `xray-ecs` branch, the Scorekeep sample application shows how to instrument an application running in Amazon Elastic Container Service (Amazon ECS). The branch provides scripts and configuration files for creating, uploading, and running Docker images in a Multicontainer Docker environment in AWS Elastic Beanstalk.

The project includes three Dockerfiles that define container images for the API, front end, and X-Ray daemon components.

- `/Dockerfile` – The Scorekeep API.
- `/scorekeep-frontend/Dockerfile` – The Angular web app client, and the nginx proxy that routes incoming traffic.
- `/xray-daemon/Dockerfile` – The X-Ray daemon.

The X-Ray daemon Dockerfile creates an image based on Amazon Linux that runs the X-Ray daemon. Download the complete example image on Docker Hub.

**Example Dockerfile – Amazon Linux**

```
FROM amazonlinux
RUN yum install -y unzip
RUN curl -o daemon.zip https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-daemon-linux-3.x.zip
RUN unzip daemon.zip && cp xray /usr/bin/xray
ENTRYPOINT ["/usr/bin/xray", "-t", "0.0.0.0:2000", "-b", "0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

The makefile in the same directory defines commands for building the image, uploading it to Amazon ECR, and running it locally (p. 143).

To run the containers on Amazon ECS, the branch includes a script to generate a `Dockerrun.aws.json` file, which you can deploy to a multicontainer Docker environment in Elastic Beanstalk. The template that the script uses shows how to write a task definition that configures networking between the containers in Amazon ECS (p. 149).

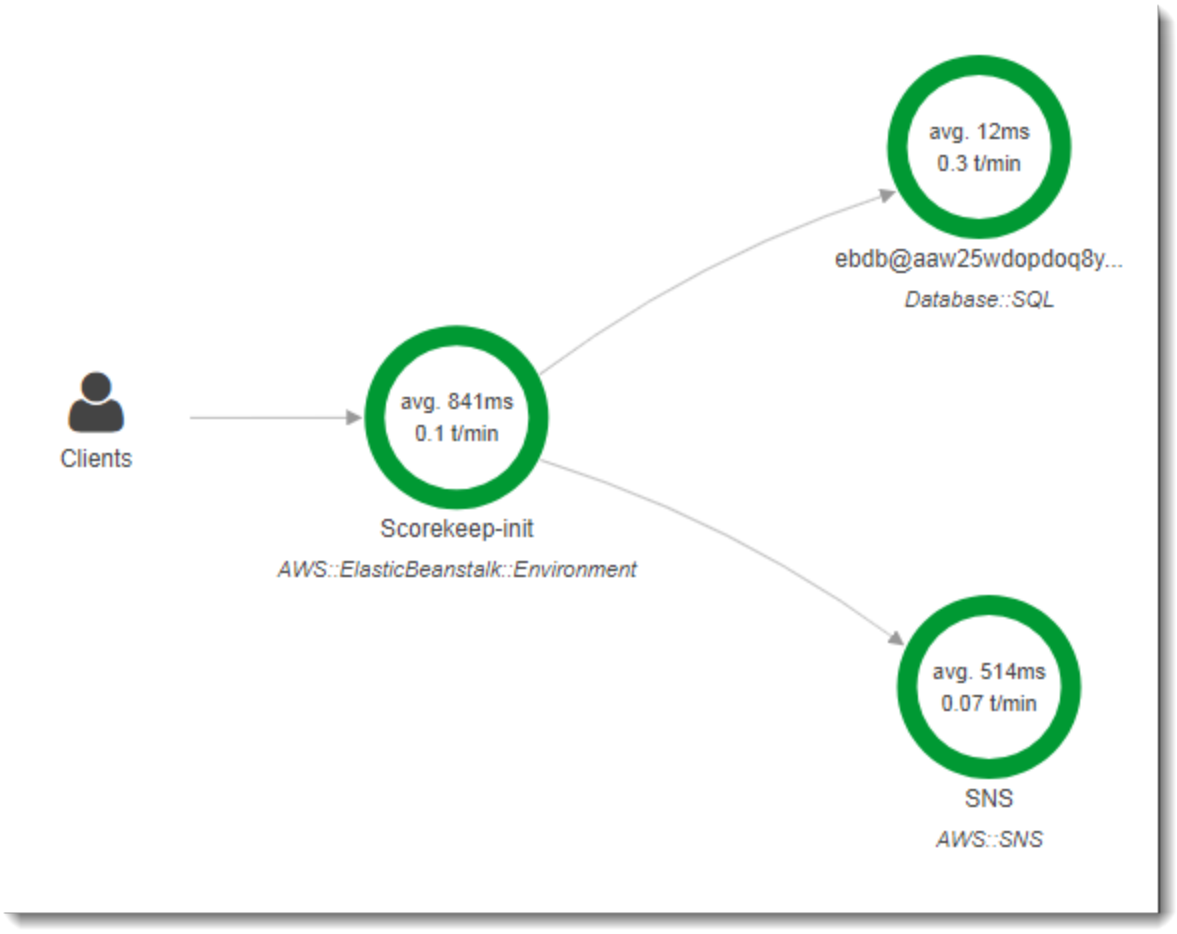
**Note**
`Dockerrun.aws.json` is the Elastic Beanstalk version of an Amazon ECS task definition file. If you don't want to use Elastic Beanstalk to create your Amazon ECS cluster, you can modify the `Dockerrun.aws.json` file to run on Amazon ECS directly by removing the `AWSEBDockerrunVersion` key from the file.

See the branch README for instructions on how to deploy Scorekeep to Amazon ECS.

# Instrumenting startup code

The X-Ray SDK for Java automatically creates segments for incoming requests. As long as a request is in scope, you can use instrumented clients and record subsegments without issue. If you try to use an instrumented client in startup code, though, you'll get a SegmentNotFoundException.

Startup code runs outside of the standard request/response flow of a web application, so you need to create segments manually to instrument it. Scorekeep shows the instrumentation of startup code in its `WebConfig` files. Scorekeep calls an SQL database and Amazon SNS during startup.




The default `WebConfig` class creates an Amazon SNS subscription for notifications. To provide a segment for the X-Ray SDK to write to when the Amazon SNS client is used, Scorekeep calls `beginSegment` and `endSegment` on the global recorder.

**Example `src/main/java/scorekeep/WebConfig.java` – Instrumented AWS SDK client in startup code**

```
AWSXRay.beginSegment("Scorekeep-init");
if ( System.getenv("NOTIFICATION_EMAIL") != null ){
  try { Sns.createSubscription(); }
  catch (Exception e ) {
    logger.warn("Failed to create subscription for email "+
 System.getenv("NOTIFICATION_EMAIL"));
  }
}
AWSXRay.endSegment();
```

In `RdsWebConfig`, which Scorekeep uses when an Amazon RDS database is connected, the configuration also creates a segment for the SQL client that Hibernate uses when it applies the database schema during startup.

**Example `src/main/java/scorekeep/RdsWebConfig.java` – Instrumented SQL database client in startup code**

```
@PostConstruct
public void schemaExport() {
  EntityManagerFactoryImpl entityManagerFactoryImpl = (EntityManagerFactoryImpl)
 localContainerEntityManagerFactoryBean.getNativeEntityManagerFactory();
  SessionFactoryImplementor sessionFactoryImplementor =
 entityManagerFactoryImpl.getSessionFactory();
  StandardServiceRegistry standardServiceRegistry =
 sessionFactoryImplementor.getSessionFactoryOptions().getServiceRegistry();
  MetadataSources metadataSources = new MetadataSources(new
 BootstrapServiceRegistryBuilder().build());
  metadataSources.addAnnotatedClass(GameHistory.class);
  MetadataImplementor metadataImplementor = (MetadataImplementor)
 metadataSources.buildMetadata(standardServiceRegistry);
  SchemaExport schemaExport = new SchemaExport(standardServiceRegistry,
 metadataImplementor);

  AWSXRay.beginSegment("Scorekeep-init");
  schemaExport.create(true, true);
  AWSXRay.endSegment();
}
```

`SchemaExport` runs automatically and uses an SQL client. Since the client is instrumented, Scorekeep must override the default implementation and provide a segment for the SDK to use when the client is invoked.

# Instrumenting scripts

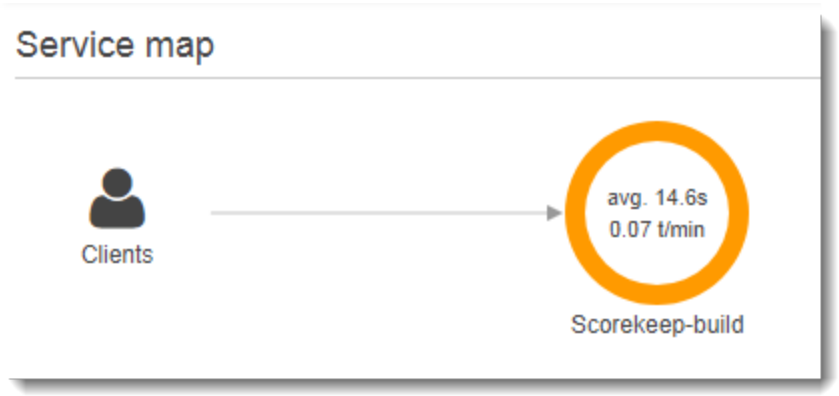
You can also instrument code that isn't part of your application. When the X-Ray daemon is running, it will relay any segments that it receives to X-Ray, even if they are not generated by the X-Ray SDK. Scorekeep uses its own scripts to instrument the build that compiles the application during deployment.

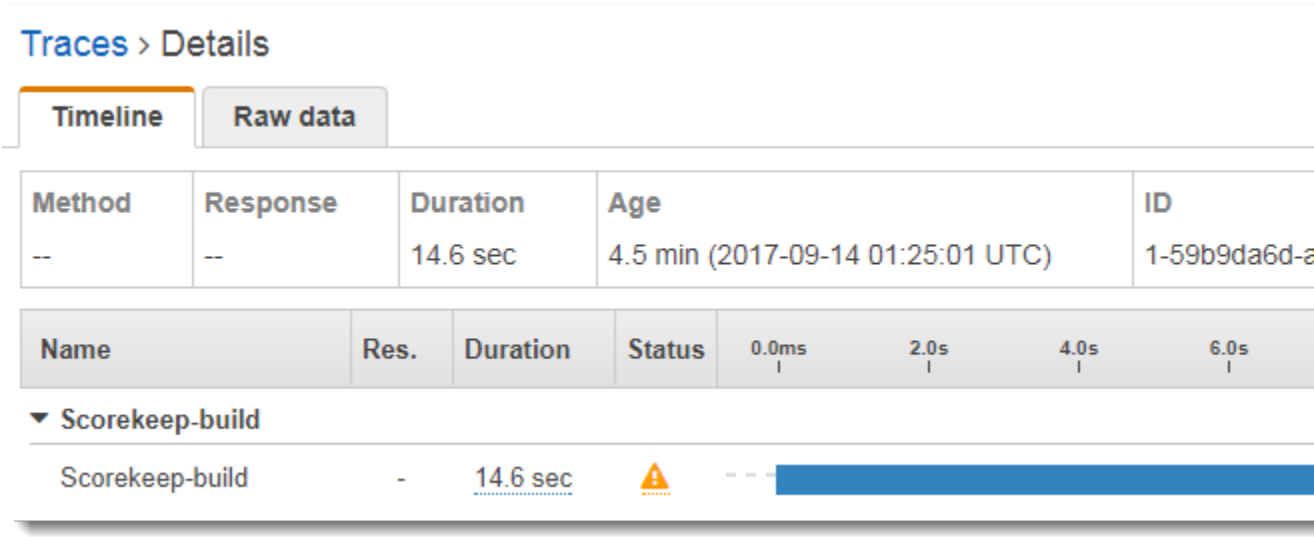
**Example `bin/build.sh` – Instrumented build script**

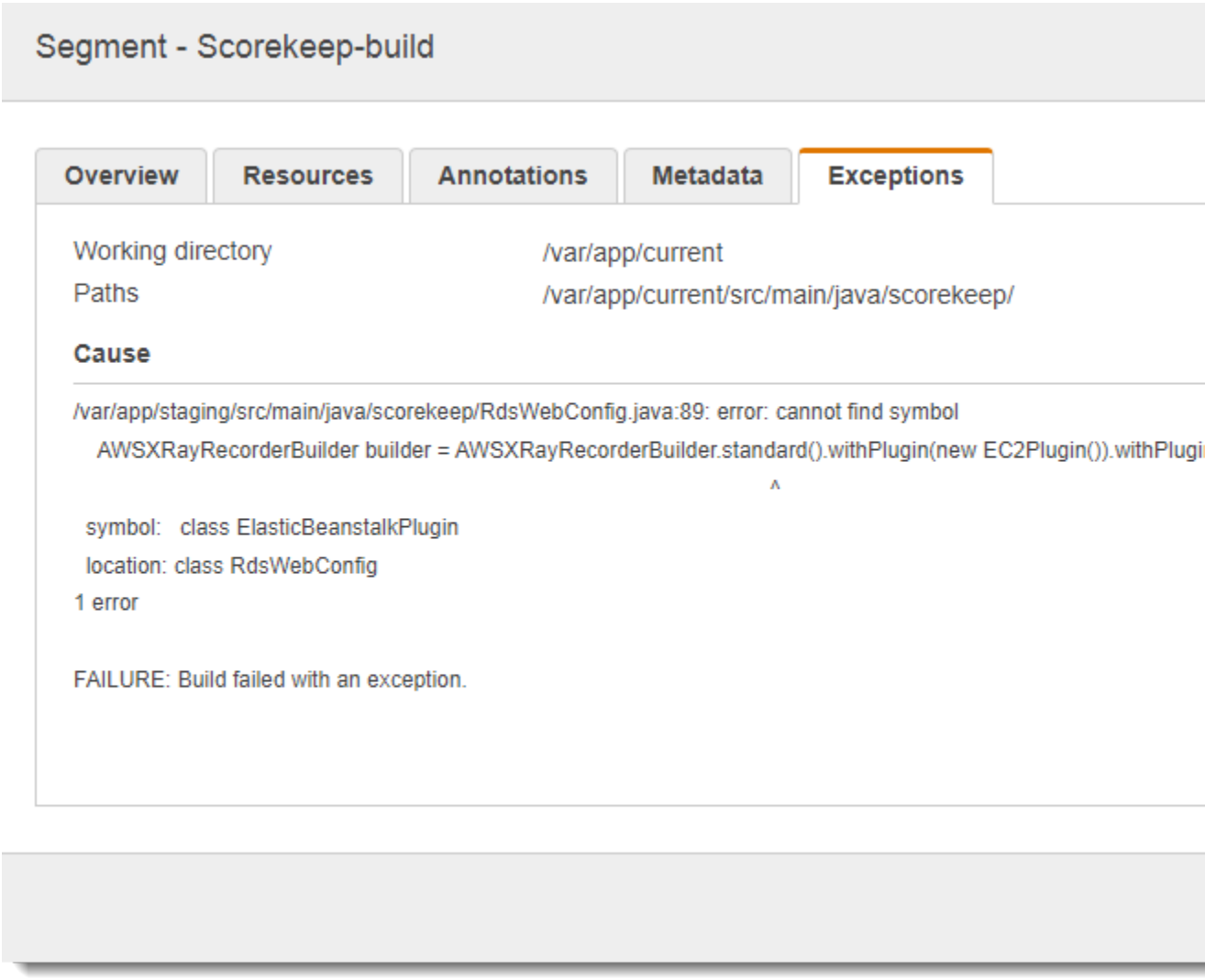
```
SEGMENT=$(python bin/xray_start.py)
gradle build --quiet --stacktrace &> /var/log/gradle.log; GRADLE_RETURN=$?
if (( GRADLE_RETURN != 0 )); then
```

```
  echo "Gradle failed with exit status $GRADLE_RETURN" >&2
  python bin/xray_error.py "$SEGMENT" "$(cat /var/log/gradle.log)"
  exit 1
fi
python bin/xray_success.py "$SEGMENT"
```

`xray_start.py`, `xray_error.py` and `xray_success.py` are simple Python scripts that construct segment objects, convert them to JSON documents, and send them to the daemon over UDP. If the Gradle build fails, you can find the error message by clicking on the **scorekeep-build** node in the X-Ray console service map.






# Instrumenting a web app client

In the `xray-cognito` branch, Scorekeep uses Amazon Cognito to enable users to create an account and sign in with it to retrieve their user information from an Amazon Cognito user pool. When a user signs in, Scorekeep uses an Amazon Cognito identity pool to get temporary AWS credentials for use with the AWS SDK for JavaScript.

The identity pool is configured to let signed-in users write trace data to AWS X-Ray. The web app uses these credentials to record the signed-in user's ID, the browser path, and the client's view of calls to the Scorekeep API.

Most of the work is done in a service class named `xray`. This service class provides methods for generating the required identifiers, creating in-progress segments, finalizing segments, and sending segment documents to the X-Ray API.

**Example `public/xray.js` – Record and upload segments**

```
...
```

```
  service.beginSegment = function() {
    var segment = {};
    var traceId = '1-' + service.getHexTime() + '-' + service.getHexId(24);

    var id = service.getHexId(16);
    var startTime = service.getEpochTime();

    segment.trace_id = traceId;
    segment.id = id;
    segment.start_time = startTime;
    segment.name = 'Scorekeep-client';
    segment.in_progress = true;
    segment.user =  sessionStorage['userid'];
    segment.http = {
      request: {
        url: window.location.href
      }
    };

    var documents = [];
    documents[0] = JSON.stringify(segment);
    service.putDocuments(documents);
    return segment;
  }

  service.endSegment = function(segment) {
    var endTime = service.getEpochTime();
    segment.end_time = endTime;
    segment.in_progress = false;
    var documents = [];
    documents[0] = JSON.stringify(segment);
    service.putDocuments(documents);
  }

  service.putDocuments = function(documents) {
    var xray = new AWS.XRay();
    var params = {
      TraceSegmentDocuments: documents
    };
    xray.putTraceSegments(params, function(err, data) {
      if (err) {
        console.log(err, err.stack);
      } else {
        console.log(data);
      }
    })
  }
```

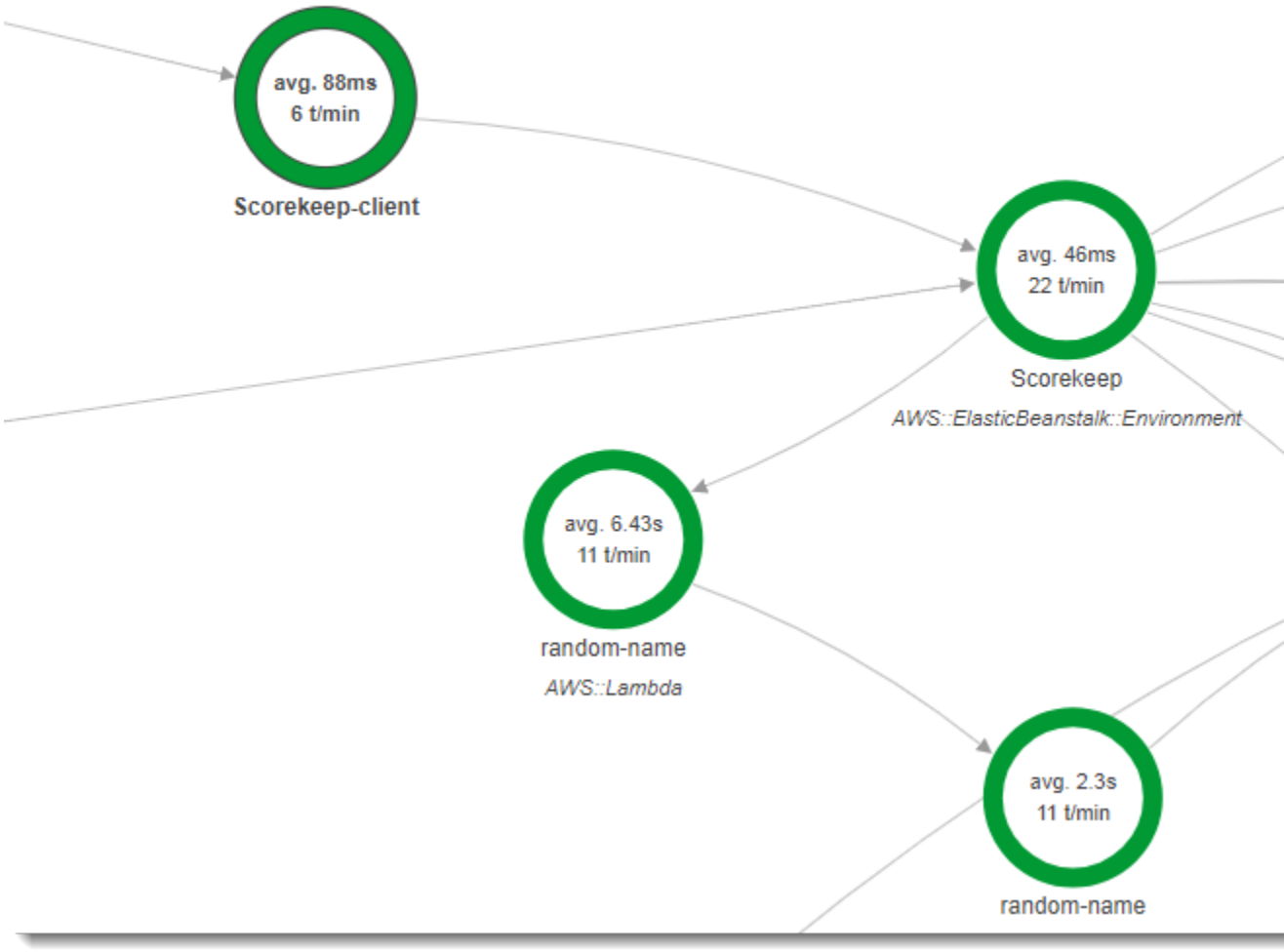
These methods are called in header and `transformResponse` functions in the resource services that the web app uses to call the Scorekeep API. To include the client segment in the same trace as the segment that the API generates, the web app must include the trace ID and segment ID in a tracing header (p. 26) (`X-Amzn-Trace-Id`) that the X-Ray SDK can read. When the instrumented Java application receives a request with this header, the X-Ray SDK for Java uses the same trace ID and makes the segment from the web app client the parent of its segment.

**Example `public/app/services.js` – Recording segments for angular resource calls and writing tracing headers**

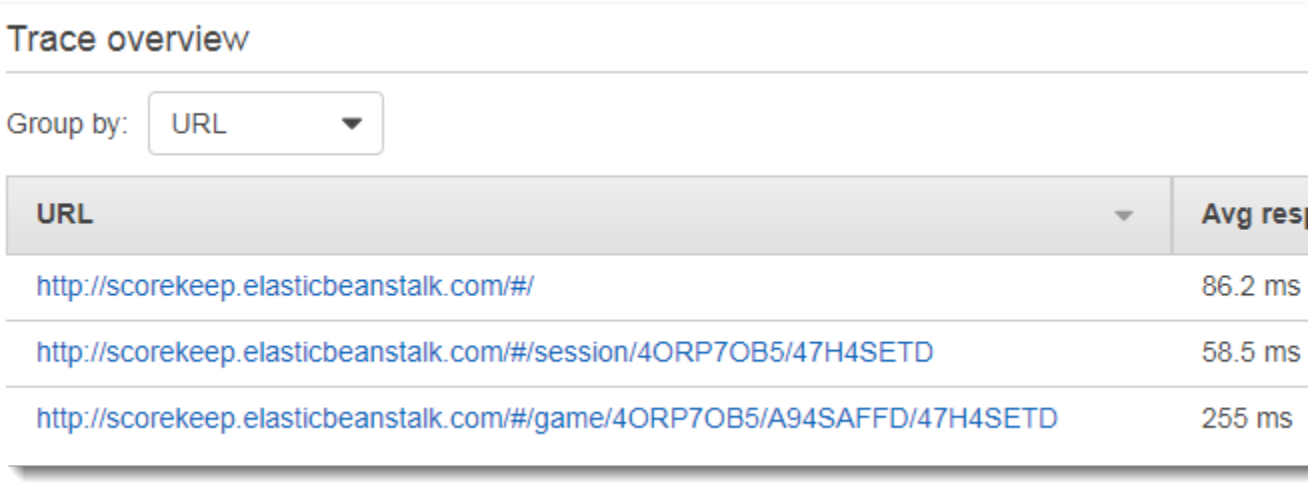
```
var module = angular.module('scorekeep');
module.factory('SessionService', function($resource, api, XRay) {
  return $resource(api + 'session/:id', { id: '@_id' }, {
    segment: {},
    get: {
```

```
      method: 'GET',
      headers: {
        'X-Amzn-Trace-Id': function(config) {
          segment = XRay.beginSegment();
          return XRay.getTraceHeader(segment);
        }
      },
      transformResponse: function(data) {
        XRay.endSegment(segment);
        return angular.fromJson(data);
      },
    },
...
```

The resulting service map includes a node for the web app client.




Traces that include segments from the web app show the URL that the user sees in the browser (paths starting with `/#/`). Without client instrumentation, you only get the URL of the API resource that the web app calls (paths starting with `/api/`).




# Using instrumented clients in worker threads

Scorekeep uses a worker thread to publish a notification to Amazon SNS when a user wins a game. Publishing the notification takes longer than the rest of the request operations combined, and doesn't affect the client or user. Therefore, performing the task asynchronously is a good way to improve response time.

However, the X-Ray SDK for Java doesn't know which segment was active when the thread is created. As a result, when you try to use the instrumented AWS SDK for Java client within the thread, it throws a `SegmentNotFoundException`, crashing the thread.

**Example Web-1.error.log**

```
Exception in thread "Thread-2" com.amazonaws.xray.exceptions.SegmentNotFoundException:
 Failed to begin subsegment named 'AmazonSNS': segment cannot be found.
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at
 sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
        at
 sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
...
```

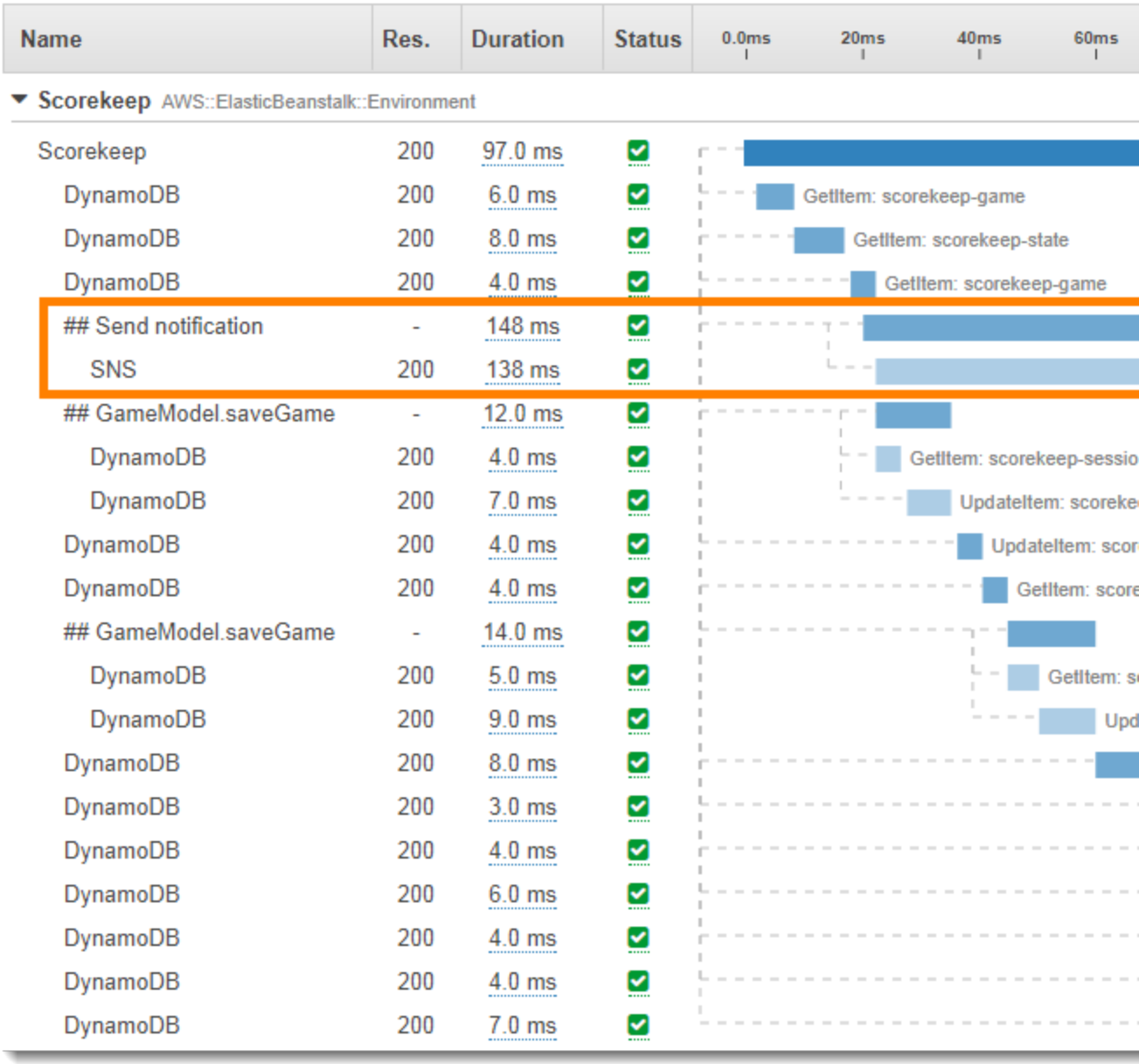
To fix this, the application uses `GetTraceEntity` to get a reference to the segment in the main thread, and `SetTraceEntity` to pass the segment back to the recorder in the worker thread.

**Example `src/main/java/scorekeep/MoveFactory.java` – Passing trace context to a worker thread**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorder;
import com.amazonaws.xray.entities.Entity;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
      Entity segment = recorder.getTraceEntity();
      Thread comm = new Thread() {
        public void run() {
```

```
        recorder.setTraceEntity(segment);
        Subsegment subsegment = AWSXRay.beginSubsegment("## Send notification");
        Sns.sendNotification("Scorekeep game completed", "Winner: " + userId);
        AWSXRay.endSubsegment();
    }
```

Because the request is now resolved before the call to Amazon SNS, the application creates a separate subsegment for the thread. This prevents the X-Ray SDK from closing the segment before it records the response from Amazon SNS. If no subsegment is open when Scorekeep resolved the request, the response from Amazon SNS could be lost.




See Passing segment context between threads in a multithreaded application (p. 210) for more information about multithreading.

# Deep linking to the X-Ray console

The session and game pages of the Scorekeep web app use deep linking to link to filtered trace lists and service maps.

**Example `public/game.html` – Deep links**

```
<div id="xray-link">
  <p><a href="https://console.aws.amazon.com/xray/home#/traces?filter=http.url%20CONTAINS
%20%22{{ gameid }}%22&timeRange=PT1H" target="blank">View traces for this game</a></p>
  <p><a href="https://console.aws.amazon.com/xray/home#/service-map&timeRange=PT1H"
 target="blank">View service map</a></p>
</div>
```

See Deep linking (p. 67) for details on how to construct deep links.

# AWS X-Ray daemon

The AWS X-Ray daemon is a software application that listens for traffic on UDP port 2000, gathers raw segment data, and relays it to the AWS X-Ray API. The daemon works in conjunction with the AWS X-Ray SDKs and must be running so that data sent by the SDKs can reach the X-Ray service.

> **Note**
> The X-Ray daemon is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-daemon

On AWS Lambda and AWS Elastic Beanstalk, use those services' integration with X-Ray to run the daemon. Lambda runs the daemon automatically any time a function is invoked for a sampled request. On Elastic Beanstalk, use the `XRayEnabled` configuration option (p. 145) to run the daemon on the instances in your environment.

To run the X-Ray daemon locally, on-premises, or on other AWS services, download it from Amazon S3 (p. 137), run it (p. 138), and then give it permission (p. 139) to upload segment documents to X-Ray.

## Downloading the daemon

You can download the daemon from Amazon S3 to run it locally, or to install it on an Amazon EC2 instance on launch.

### X-Ray daemon installers and executables

- **Linux (executable)** – `aws-xray-daemon-linux-3.x.zip` (sig)
- **Linux (RPM installer)** – `aws-xray-daemon-3.x.rpm`
- **Linux (DEB installer)** – `aws-xray-daemon-3.x.deb`
- **OS X (executable)** – `aws-xray-daemon-macos-3.x.zip` (sig)
- **Windows (executable)** – `aws-xray-daemon-windows-process-3.x.zip` (sig)
- **Windows (service)** – `aws-xray-daemon-windows-service-3.x.zip` (sig)

These links always point to the latest release of the daemon. To download a specific release, replace `3.x` with the version number. For example, `2.1.0`.

X-Ray assets are replicated to buckets in every supported region. To use the bucket closest to you or your AWS resources, replace the region in the above links with your region.

```
https://s3.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-daemon/aws-xray-daemon-3.x.rpm
```

## Verifying the daemon archive's signature

GPG signature files are included for daemon assets compressed in ZIP archives. The public key is here: `aws-xray.gpg`.

You can use the public key to verify that the daemon's ZIP archive is original and unmodified. First, import the public key with GnuPG.

**To import the public key**

1. Download the public key.

```
$ BUCKETURL=https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2
$ wget $BUCKETURL/xray-daemon/aws-xray.gpg
```

2. Import the public key into your keyring.

```
$ gpg --import aws-xray.gpg
gpg: /Users/me/.gnupg/trustdb.gpg: trustdb created
gpg: key 7BFE036BFE6157D3: public key "AWS X-Ray <aws-xray@amazon.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1
```

Use the imported key to verify the signature of the daemon's ZIP archive.

**To verify an archive's signature**

1. Download the archive and signature file.

```
$ BUCKETURL=https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2
$ wget $BUCKETURL/xray-daemon/aws-xray-daemon-linux-3.0.0.zip
$ wget $BUCKETURL/xray-daemon/aws-xray-daemon-linux-3.0.0.zip.sig
```

2. Run `gpg --verify` to verify the signature.

```
$ gpg --verify aws-xray-daemon-linux-3.0.0.zip.sig aws-xray-daemon-linux-3.0.0.zip
gpg: Signature made Wed 19 Apr 2017 05:06:31 AM UTC using RSA key ID FE6157D3
gpg: Good signature from "AWS X-Ray <aws-xray@amazon.com>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
Primary key fingerprint: EA6D 9271 FBF3 6990 277F  4B87 7BFE 036B FE61 57D3
```

Note the warning about trust. A key is only trusted if you or someone you trust has signed it. This does not mean that the signature is invalid, only that you have not verified the public key.

# Running the daemon

Run the daemon locally from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
~/Downloads$ ./xray -o -n us-east-2
```

For detailed platform-specific instructions, see the following topics:

- **Linux (local)** – Running the X-Ray daemon on Linux (p. 142)
- **Windows (local)** – Running the X-Ray daemon on Windows (p. 144)
- **Elastic Beanstalk** – Running the X-Ray daemon on AWS Elastic Beanstalk (p. 145)
- **Amazon EC2** – Running the X-Ray daemon on Amazon EC2 (p. 148)

- **Amazon ECS** – Running the X-Ray daemon on Amazon ECS (p. 149)

You can customize the daemon's behavior further by using command line options or a configuration file. See Configuring the AWS X-Ray daemon (p. 140) for details.

# Giving the daemon permission to send data to X-Ray

The X-Ray daemon uses the AWS SDK to upload trace data to X-Ray, and it needs AWS credentials with permission to do that.

On Amazon EC2, the daemon uses the instance's instance profile role automatically. Locally, save your access keys to a file named `credentials` in your user directory under a folder named `.aws`.

**Example ~/.aws/credentials**

```
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

If you specify credentials in more than one location (credentials file, instance profile, or environment variables), the SDK provider chain determines which credentials are used. For more information about providing credentials to the SDK, see Specifying Credentials in the *AWS SDK for Go Developer Guide*.

The IAM role or user that the daemon's credentials belong to must have permission to write data to the service on your behalf.

- To use the daemon on Amazon EC2, create a new instance profile role or add the managed policy to an existing one.
- To use the daemon on Elastic Beanstalk, add the managed policy to the Elastic Beanstalk default instance profile role.
- To run the daemon locally, create an IAM user and save its access keys on your computer.

For more information, see Identity and access management for AWS X-Ray (p. 30).

# X-Ray daemon logs

The daemon outputs information about its current configuration and segments that it sends to AWS X-Ray.

```
2016-11-24T06:07:06Z [Info] Initializing AWS X-Ray daemon 2.1.0
2016-11-24T06:07:06Z [Info] Using memory limit of 49 MB
2016-11-24T06:07:06Z [Info] 313 segment buffers allocated
2016-11-24T06:07:08Z [Info] Successfully sent batch of 1 segments (0.123 seconds)
2016-11-24T06:07:09Z [Info] Successfully sent batch of 1 segments (0.006 seconds)
```

By default, the daemon outputs logs to STDOUT. If you run the daemon in the background, use the `--log-file` command line option or a configuration file to set the log file path. You can also set the log level and disable log rotation. See Configuring the AWS X-Ray daemon (p. 140) for instructions.

On Elastic Beanstalk, the platform sets the location of the daemon logs. See Running the X-Ray daemon on AWS Elastic Beanstalk (p. 145) for details.

# Configuring the AWS X-Ray daemon

You can use command line options or a configuration file to customize the X-Ray daemon's behavior. Most options are available using both methods, but some are only available in configuration files and some only at the command line.

To get started, the only option that you need to know is `-n` or `--region`, which you use to set the region that the daemon uses to send trace data to X-Ray.

```
~/xray-daemon$ ./xray -n us-east-2
```

If you are running the daemon locally, that is, not on Amazon EC2, you can add the `-o` option to skip checking for instance profile credentials so the daemon will become ready more quickly.

```
~/xray-daemon$ ./xray -o -n us-east-2
```

The rest of the command line options let you configure logging, listen on a different port, limit the amount of memory that the daemon can use, or assume a role to send trace data to a different account.

You can pass a configuration file to the daemon to access advanced configuration options and do things like limit the number of concurrent calls to X-Ray, disable log rotation, and send traffic to a proxy.

**Sections**



## Supported environment variables

The X-Ray daemon supports the following environment variables:

- `AWS_REGION` – Specifies the AWS Region of the X-Ray service endpoint.
- `HTTPS_PROXY` – Specifies a proxy address for the daemon to upload segments through. This can be either the DNS domain names or IP addresses and port numbers used by your proxy servers.

## Using command line options

Pass these options to the daemon when you run it locally or with a user data script.

**Command Line Options**

- `-b`, `--bind` – Listen for segment documents on a different UDP port.

  ```
  --bind "127.0.0.1:3000"
  ```

  Default – `2000`.
- `-t`, `--bind-tcp` – Listen for calls to the X-Ray service on a different TCP port.

```
-bind-tcp "127.0.0.1:3000"
```

Default – 2000.

- -c, --config – Load a configuration file from the specified path.

```
--config "/home/ec2-user/xray-daemon.yaml"
```

- -f, --log-file – Output logs to the specified file path.

```
--log-file "/var/log/xray-daemon.log"
```

- -l, --log-level – Log level, from most verbose to least: dev, debug, info, warn, error, prod.

```
--log-level warn
```

Default – prod

- -m, --buffer-memory – Change the amount of memory in MB that buffers can use (minimum 3).

```
--buffer-memory 50
```

Default – 1% of available memory.

- -o, --local-mode – Don't check for EC2 instance metadata.
- -r, --role-arn – Assume the specified IAM role to upload segments to a different account.

```
--role-arn "arn:aws:iam::123456789012:role/xray-cross-account"
```

- -a, --resource-arn – Amazon Resource Name (ARN) of the AWS resource running the daemon.
- -p, --proxy-address – Upload segments to AWS X-Ray through a proxy.
- -n, --region – Send segments to X-Ray service in a specific region.
- -v, --version – Show AWS X-Ray daemon version.
- -h, --help – Show the help screen.

# Using a configuration file

You can also use a YAML format file to configure the daemon. Pass the configuration file to the daemon by using the -c option.

```
~$ ./xray -c ~/xray-daemon.yaml
```

### Configuration file options

- TotalBufferSizeMB – Maximum buffer size in MB (minimum 3). Choose 0 to use 1% of host memory.
- Concurrency – Maximum number of concurrent calls to AWS X-Ray to upload segment documents.
- Region – Send segments to AWS X-Ray service in a specific region.
- Socket – Configure the daemon's binding.
  - UDPAddress – Change the port on which the daemon listens.
  - TCPAddress – Listen for calls to the X-Ray service (p. 100) on a different TCP port.

- `Logging` – Configure logging behavior.
  - `LogRotation` – Set to `false` to disable log rotation.
  - `LogLevel` – Change the log level, from most verbose to least: `dev`, `debug`, `info`, `warn`, `error`, `prod` (default).
  - `LogPath` – Output logs to the specified file path.
- `LocalMode` – Set to `true` to skip checking for EC2 instance metadata.
- `ResourceARN` – Amazon Resource Name (ARN) of the AWS resource running the daemon.
- `RoleARN` – Assume the specified IAM role to upload segments to a different account.
- `ProxyAddress` – Upload segments to AWS X-Ray through a proxy.
- `Endpoint` – Change the X-Ray service endpoint to which the daemon sends segment documents.
- `NoVerifySSL` – Disable TLS certificate verification.
- `Version` – Daemon configuration file format version.

**Example Xray-daemon.yaml**

This configuration file changes the daemon's listening port to 3000, turns off checks for instance metadata, sets a role to use for uploading segments, and changes region and logging options.

```
Socket:
  UDPAddress: "127.0.0.1:3000"
  TCPAddress: "127.0.0.1:3000"
Region: "us-west-2"
Logging:
  LogLevel: "warn"
  LogPath: "/var/log/xray-daemon.log"
LocalMode: true
RoleARN: "arn:aws:iam::123456789012:role/xray-cross-account"
Version: 2
```

# Running the X-Ray daemon locally

You can run the AWS X-Ray daemon locally on Linux, MacOS, Windows, or in a Docker container. Run the daemon to relay trace data to X-Ray when you are developing and testing your instrumented application. Download and extract the daemon by using the instructions here (p. 137).

When running locally, the daemon can read credentials from an AWS SDK credentials file (`.aws/credentials` in your user directory) or from environment variables. For more information, see Giving the daemon permission to send data to X-Ray (p. 139).

The daemon listens for UDP data on port 2000. You can change the port and other options by using a configuration file and command line options. For more information, see Configuring the AWS X-Ray daemon (p. 140).

## Running the X-Ray daemon on Linux

You can run the daemon executable from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
~/xray-daemon$ ./xray -o -n us-east-2
```

To run the daemon in the background, use `&`.

```
~/xray-daemon$ ./xray -o -n us-east-2 &
```

Terminate a daemon process running in the background with `pkill`.

```
~$ pkill xray
```

# Running the X-Ray daemon in a Docker container

To run the daemon locally in a Docker container, save the following text to a file named `Dockerfile`. Download the complete example image on Docker Hub.

### Example Dockerfile – Amazon Linux

```
FROM amazonlinux
RUN yum install -y unzip
RUN curl -o daemon.zip https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-daemon-linux-3.x.zip
RUN unzip daemon.zip && cp xray /usr/bin/xray
ENTRYPOINT ["/usr/bin/xray", "-t", "0.0.0.0:2000", "-b", "0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

Build the container image with `docker build`.

```
~/xray-daemon$ docker build -t xray-daemon .
```

Run the image in a container with `docker run`.

```
~/xray-daemon$ docker run \
      --attach STDOUT \
      -v ~/.aws/:/root/.aws/:ro \
      --net=host \
      -e AWS_REGION=us-east-2 \
      --name xray-daemon \
      -p 2000:2000/udp \
      xray-daemon -o
```

This command uses the following options:

- `--attach STDOUT` – View output from the daemon in the terminal.
- `-v ~/.aws/:/root/.aws/:ro` – Give the container read-only access to the `.aws` directory to let it read your AWS SDK credentials.
- `AWS_REGION=us-east-2` – Set the `AWS_REGION` environment variable to tell the daemon which region to use.
- `--net=host` – Attach the container to the `host` network. Containers on the host network can communicate with each other without publishing ports.
- `-p 2000:2000/udp` – Map UDP port 2000 on your machine to the same port on the container. This is not required for containers on the same network to communicate, but it does let you send segments to the daemon from the command line (p. 85) or from an application not running in Docker.
- `--name xray-daemon` – Name the container `xray-daemon` instead of generating a random name.
- `-o` (after the image name) – Append the `-o` option to the entry point that runs the daemon within the container. This option tells the daemon to run in local mode to prevent it from trying to read Amazon EC2 instance metadata.

To stop the daemon, use `docker stop`. If you make changes to the `Dockerfile` and build a new image, you need to delete the existing container before you can create another one with the same name. Use `docker rm` to delete the container.

```
$ docker stop xray-daemon
$ docker rm xray-daemon
```

The Scorekeep sample application shows how to use the X-Ray daemon in a local Docker container. See Instrumenting Amazon ECS applications (p. 127) for details.

# Running the X-Ray daemon on Windows

You can run the daemon executable from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
> .\xray_windows.exe -o -n us-east-2
```

Use a PowerShell script to create and run a service for the daemon.

**Example PowerShell script - Windows**

```
if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ){
    sc.exe stop AWSXRayDaemon
    sc.exe delete AWSXRayDaemon
}
if ( Get-Item -path aws-xray-daemon -ErrorAction SilentlyContinue ) {
    Remove-Item -Recurse -Force aws-xray-daemon
}

$currentLocation = Get-Location
$zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
$zipPath = "$currentLocation\$zipFileName"
$destPath = "$currentLocation\aws-xray-daemon"
$daemonPath = "$destPath\xray.exe"
$daemonLogPath = "C:\inetpub\wwwroot\xray-daemon.log"
$url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-daemon/
aws-xray-daemon-windows-service-3.x.zip"

Invoke-WebRequest -Uri $url -OutFile $zipPath
Add-Type -Assembly "System.IO.Compression.Filesystem"
[io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

sc.exe create AWSXRayDaemon binPath= "$daemonPath -f $daemonLogPath"
sc.exe start AWSXRayDaemon
```

# Running the X-Ray daemon on OS X

You can run the daemon executable from the command line. Use the `-o` option to run in local mode, and `-n` to set the region.

```
~/xray-daemon$ ./xray_mac -o -n us-east-2
```

To run the daemon in the background, use `&`.

```
~/xray-daemon$ ./xray_mac -o -n us-east-2 &
```

Use `nohup` to prevent the daemon from terminating when the terminal is closed.

```
~/xray-daemon$ nohup ./xray_mac &
```

# Running the X-Ray daemon on AWS Elastic Beanstalk

To relay trace data from your application to AWS X-Ray, you can run the X-Ray daemon on your Elastic Beanstalk environment's Amazon EC2 instances. For a list of supported platforms, see Configuring AWS X-Ray Debugging in the *AWS Elastic Beanstalk Developer Guide*.

> **Note**
> The daemon uses your environment's instance profile for permissions. For instructions about adding permissions to the Elastic Beanstalk instance profile, see Giving the daemon permission to send data to X-Ray (p. 139).

Elastic Beanstalk platforms provide a configuration option that you can set to run the daemon automatically. You can enable the daemon in a configuration file in your source code or by choosing an option in the Elastic Beanstalk console. When you enable the configuration option, the daemon is installed on the instance and runs as a service.

The version included on Elastic Beanstalk platforms might not be the latest version. See the Supported Platforms topic to find out the version of the daemon that is available for your platform configuration.

Elastic Beanstalk does not provide the X-Ray daemon on the Multicontainer Docker (Amazon ECS) platform. The Scorekeep sample application shows how to use the X-Ray daemon on Amazon ECS with Elastic Beanstalk. See Instrumenting Amazon ECS applications (p. 127) for details.

## Using the Elastic Beanstalk X-Ray integration to run the X-Ray daemon

Use the console to turn on X-Ray integration, or configure it in your application source code with a configuration file.

### To enable the X-Ray daemon in the Elastic Beanstalk console

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Choose **Configuration**.
4. Choose **Software Settings**.
5. For **X-Ray daemon**, choose **Enabled**.
6. Choose **Apply**.

You can include a configuration file in your source code to make your configuration portable between environments.

### Example .ebextensions/xray-daemon.config

```
option_settings:
  aws:elasticbeanstalk:xray:
    XRayEnabled: true
```

Elastic Beanstalk passes a configuration file to the daemon and outputs logs to a standard location.

**On Windows Server Platforms**

- **Configuration file** – `C:\Progam Files\Amazon\XRay\cfg.yaml`
- **Logs** – `c:\Program Files\Amazon\XRay\logs\xray-service.log`

**On Linux Platforms**

- **Configuration file** – `/etc/amazon/xray/cfg.yaml`
- **Logs** – `/var/log/xray/xray.log`

Elastic Beanstalk provides tools for pulling instance logs from the AWS Management Console or command line. You can tell Elastic Beanstalk to include the X-Ray daemon logs by adding a task with a configuration file.

**Example .ebextensions/xray-logs.config - Linux**

```
files:
  "/opt/elasticbeanstalk/tasks/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      /var/log/xray/xray.log
```

**Example .ebextensions/xray-logs.config - Windows server**

```
files:
  "c:/Program Files/Amazon/ElasticBeanstalk/config/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      c:\Progam Files\Amazon\XRay\logs\xray-service.log
```

See Viewing Logs from Your Elastic Beanstalk Environment's Amazon EC2 Instances in the *AWS Elastic Beanstalk Developer Guide* for more information.

# Downloading and running the X-Ray daemon manually (advanced)

If the X-Ray daemon isn't available for your platform configuration, you can download it from Amazon S3 and run it with a configuration file.

Use an Elastic Beanstalk configuration file to download and run the daemon.

**Example .ebextensions/xray.config - Linux**

```
commands:
  01-stop-tracing:
    command: yum remove -y xray
    ignoreErrors: true
  02-copy-tracing:
    command: curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/
aws-xray-daemon-3.x.rpm -o /home/ec2-user/xray.rpm
```

```
  03-start-tracing:
    command: yum install -y /home/ec2-user/xray.rpm

files:
  "/opt/elasticbeanstalk/tasks/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      /var/log/xray/xray.log
  "/etc/amazon/xray/cfg.yaml" :
    mode: "000644"
    owner: root
    group: root
    content: |
      Logging:
        LogLevel: "debug"
```

**Example .ebextensions/xray.config - Windows server**

```
container_commands:
  01-execute-config-script:
    command: Powershell.exe -ExecutionPolicy Bypass -File c:\\temp\\installDaemon.ps1
    waitAfterCompletion: 0

files:
  "c:/temp/installDaemon.ps1":
    content: |
      if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ) {
          sc.exe stop AWSXRayDaemon
          sc.exe delete AWSXRayDaemon
      }

      $targetLocation = "C:\Program Files\Amazon\XRay"
      if ((Test-Path $targetLocation) -eq 0) {
          mkdir $targetLocation
      }

      $zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
      $zipPath = "$targetLocation\$zipFileName"
      $destPath = "$targetLocation\aws-xray-daemon"
      if ((Test-Path $destPath) -eq 1) {
          Remove-Item -Recurse -Force $destPath
      }

      $daemonPath = "$destPath\xray.exe"
      $daemonLogPath = "$targetLocation\xray-daemon.log"
      $url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-
daemon/aws-xray-daemon-windows-service-3.x.zip"

      Invoke-WebRequest -Uri $url -OutFile $zipPath
      Add-Type -Assembly "System.IO.Compression.Filesystem"
      [io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

      New-Service -Name "AWSXRayDaemon" -StartupType Automatic -BinaryPathName
 "`"$daemonPath`" -f `"$daemonLogPath`""
      sc.exe start AWSXRayDaemon
    encoding: plain
  "c:/Program Files/Amazon/ElasticBeanstalk/config/taillogs.d/xray-daemon.conf" :
    mode: "000644"
    owner: root
    group: root
    content: |
      C:\Program Files\Amazon\XRay\xray-daemon.log
```

These examples also add the daemon's log file to the Elastic Beanstalk tail logs task, so that it's included when you request logs with the console or Elastic Beanstalk Command Line Interface (EB CLI).

# Running the X-Ray daemon on Amazon EC2

You can run the X-Ray daemon on the following operating systems on Amazon EC2:

- Amazon Linux
- Ubuntu
- Windows Server (2012 R2 and newer)

Use an instance profile to grant the daemon permission to upload trace data to X-Ray. For more information, see Giving the daemon permission to send data to X-Ray (p. 139).

Use a user data script to run the daemon automatically when you launch the instance.

**Example User data script - Linux**

```
#!/bin/bash
curl https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-
daemon-3.x.rpm -o /home/ec2-user/xray.rpm
yum install -y /home/ec2-user/xray.rpm
```

**Example User data script - Windows server**

```
<powershell>
if ( Get-Service "AWSXRayDaemon" -ErrorAction SilentlyContinue ) {
    sc.exe stop AWSXRayDaemon
    sc.exe delete AWSXRayDaemon
}

$targetLocation = "C:\Program Files\Amazon\XRay"
if ((Test-Path $targetLocation) -eq 0) {
    mkdir $targetLocation
}

$zipFileName = "aws-xray-daemon-windows-service-3.x.zip"
$zipPath = "$targetLocation\$zipFileName"
$destPath = "$targetLocation\aws-xray-daemon"
if ((Test-Path $destPath) -eq 1) {
    Remove-Item -Recurse -Force $destPath
}

$daemonPath = "$destPath\xray.exe"
$daemonLogPath = "$targetLocation\xray-daemon.log"
$url = "https://s3.dualstack.us-west-2.amazonaws.com/aws-xray-assets.us-west-2/xray-daemon/
aws-xray-daemon-windows-service-3.x.zip"

Invoke-WebRequest -Uri $url -OutFile $zipPath
Add-Type -Assembly "System.IO.Compression.Filesystem"
[io.compression.zipfile]::ExtractToDirectory($zipPath, $destPath)

New-Service -Name "AWSXRayDaemon" -StartupType Automatic -BinaryPathName "`"$daemonPath`" -
f `"$daemonLogPath`""
sc.exe start AWSXRayDaemon
</powershell>
```

# Running the X-Ray daemon on Amazon ECS

In Amazon ECS, create a Docker image that runs the X-Ray daemon, upload it to a Docker image repository, and then deploy it to your Amazon ECS cluster. You can use port mappings and network mode settings in your task definition file to allow your application to communicate with the daemon container.

## Using the official Docker image

X-Ray provides a Docker container image that you can deploy alongside your application.

```
$ docker pull amazon/aws-xray-daemon
```

**Example Task definition**

```
{
  "name": "xray-daemon",
  "image": "amazon/aws-xray-daemon",
  "cpu": 32,
  "memoryReservation": 256,
  "portMappings" : [
      {
          "hostPort": 0,
          "containerPort": 2000,
          "protocol": "udp"
      }
   ],
}
```

## Create and build a Docker image

For custom configuration, you may need to define your own Docker image.

> **Note**
> The Scorekeep sample application shows how to use the X-Ray daemon on Amazon ECS. See Instrumenting Amazon ECS applications (p. 127) for details.

Add managed policies to your task role to grant the daemon permission to upload trace data to X-Ray. For more information, see Giving the daemon permission to send data to X-Ray (p. 139).

Use one of the following Dockerfiles to create an image that runs the daemon.

**Example Dockerfile – Amazon Linux**

```
FROM amazonlinux
RUN yum install -y unzip
RUN curl -o daemon.zip https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-daemon-linux-3.x.zip
RUN unzip daemon.zip && cp xray /usr/bin/xray
ENTRYPOINT ["/usr/bin/xray", "-t", "0.0.0.0:2000", "-b", "0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

> **Note**
> Flags `-t` and `-b` are required to specify a binding address to listen to the loopback of a multi-container environment.

### Example Dockerfile – Ubuntu

For Debian derivatives, you also need to install certificate authority (CA) certificates to avoid issues when downloading the installer.

```
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y --force-yes --no-install-recommends apt-transport-
https curl ca-certificates wget && apt-get clean && apt-get autoremove && rm -rf /var/lib/
apt/lists/*
RUN wget https://s3.us-east-2.amazonaws.com/aws-xray-assets.us-east-2/xray-daemon/aws-xray-
daemon-3.x.deb
RUN dpkg -i aws-xray-daemon-3.x.deb
ENTRYPOINT ["/usr/bin/xray", "--bind=0.0.0.0:2000", "--bind-tcp=0.0.0.0:2000"]
EXPOSE 2000/udp
EXPOSE 2000/tcp
```

In your task definition, the configuration depends on the networking mode that you use. Bridge networking is the default and can be used in your default VPC. In a bridge network, set the `AWS_XRAY_DAEMON_ADDRESS` environment variable to tell the X-Ray SDK which container-port to reference and set the host port. For example, you could publish UDP port 2000, and create a link from your application container to the daemon container.

### Example Task definition

```
    {
      "name": "xray-daemon",
      "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/xray-daemon",
      "cpu": 32,
      "memoryReservation": 256,
      "portMappings" : [
          {
              "hostPort": 0,
              "containerPort": 2000,
              "protocol": "udp"
          }
       ],
    },
    {
      "name": "scorekeep-api",
      "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/scorekeep-api",
      "cpu": 192,
      "memoryReservation": 512,
      "environment": [
          { "name" : "AWS_REGION", "value" : "us-east-2" },
          { "name" : "NOTIFICATION_TOPIC", "value" : "arn:aws:sns:us-
east-2:123456789012:scorekeep-notifications" },
          { "name" : "AWS_XRAY_DAEMON_ADDRESS", "value" : "xray-daemon:2000" }
      ],
      "portMappings" : [
          {
              "hostPort": 5000,
              "containerPort": 5000
          }
      ],
      "links": [
        "xray-daemon"
      ]
    }
```

If you run your cluster in the private subnet of a VPC, you can use the `awsvpc` network mode to attach an elastic network interface (ENI) to your containers. This enables you to avoid using links. Omit the host port in the port mappings, the link, and the `AWS_XRAY_DAEMON_ADDRESS` environment variable.

**Example VPC task definition**

```
{
    "family": "scorekeep",
    "networkMode":"awsvpc",
    "containerDefinitions": [
        {
          "name": "xray-daemon",
          "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/xray-daemon",
          "cpu": 32,
          "memoryReservation": 256,
          "portMappings" : [
              {
                  "containerPort": 2000,
                  "protocol": "udp"
              }
          ]
        },
        {
            "name": "scorekeep-api",
            "image": "123456789012.dkr.ecr.us-east-2.amazonaws.com/scorekeep-api",
            "cpu": 192,
            "memoryReservation": 512,
            "environment": [
                { "name" : "AWS_REGION", "value" : "us-east-2" },
                { "name" : "NOTIFICATION_TOPIC", "value" : "arn:aws:sns:us-
east-2:123456789012:scorekeep-notifications" }
            ],
            "portMappings" : [
                {
                    "containerPort": 5000
                }
            ]
        }
    ]
}
```

# Configure command line options in the Amazon ECS console

Command line options override any conflicting values in your image's config file. Command line options are typically used for local testing, but can also be used for convenience while setting environment variables, or to control the startup process.

By adding command line options, you are updating the Docker `CMD` that is passed to the container. For more information, see the Docker run reference.

**To set a command line option**

1. Open the Amazon ECS console at https://console.aws.amazon.com/ecs/.
2. From the navigation bar, choose the region that contains your task definition.
3. In the navigation pane, choose **Task Definitions**.
4. On the **Task Definitions** page, select the box to the left of the task definition to revise and choose **Create new revision**.
5. On the **Create new revision of Task Definition** page, select the container.

6. In the **ENVIRONMENT** section, add your comma-separated list of command line options to the **Command** field.
7. Choose **Update**.
8. Verify the information and choose **Create**.

The following example shows how to write a comma-separated command line option for the `RoleARN` option. The `RoleARN` option assumes the specified IAM role to upload segments to a different account.

### Example

```
--role-arn, arn:aws:iam::123456789012:role/xray-cross-account
```

To learn more about the available command line options in X-Ray, see Configuring the AWS X-Ray Daemon (p. 140).

# Integrating AWS X-Ray with other AWS services

Other AWS services provide integration with AWS X-Ray by adding a tracing header to requests, running the X-Ray daemon, or making sampling decisions and uploading trace data to X-Ray.

> **Note**
> The X-Ray SDKs include plugins for additional integration with AWS services. For example, you can use the X-Ray SDK for Java's Elastic Beanstalk plugin to add information about the Elastic Beanstalk environment that runs your application including the environment name and ID.

**Topics**



## Amazon API Gateway active tracing support for AWS X-Ray

Amazon API Gateway provides active tracing (p. 6) support for AWS X-Ray. Enable active tracing on your API stages to sample incoming requests and send traces to X-Ray.

**To enable active tracing on an API stage**

1. Open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. Choose an API.
3. Choose a stage.
4. On the **Logs/Tracing** tab, choose **Enable X-Ray Tracing**.
5. Choose **Resources** in the left side navigation panel.
6. To redeploy the API with the new settings, choose **Actions**, **Deploy API**.

API Gateway uses sampling rules that you define in the X-Ray console to determine which requests to record. You can create rules that only apply to APIs, or that apply only to requests that contain certain headers. API Gateway records headers in attributes on the segment, along with details about the stage and request. For more information, see Configuring sampling rules in the X-Ray console (p. 70).

For all incoming requests, API Gateway adds a tracing header (p. 26) to incoming HTTP requests that don't already have one.

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793
```

### Trace ID Format

A `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

- The version number, that is, `1`.
- The time of the original request, in Unix epoch time, in **8 hexadecimal digits**.

  For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal digits.
- A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.

If active tracing is disabled, the stage still records a segment if the request comes from a service that sampled the request and started a trace. For example, an instrumented web application can call an API Gateway API with an HTTP client. When you instrument an HTTP client with the X-Ray SDK, it adds a tracing header to the outgoing request that contains the sampling decision. API Gateway reads the tracing header and creates a segment for sampled requests.

If you use API Gateway to generate a Java SDK for your API, you can instrument the SDK client by adding a request handler with the client builder, in the same way that you would manually instrument an AWS SDK client. See Tracing AWS SDK calls with the X-Ray SDK for Java (p. 199) for instructions.

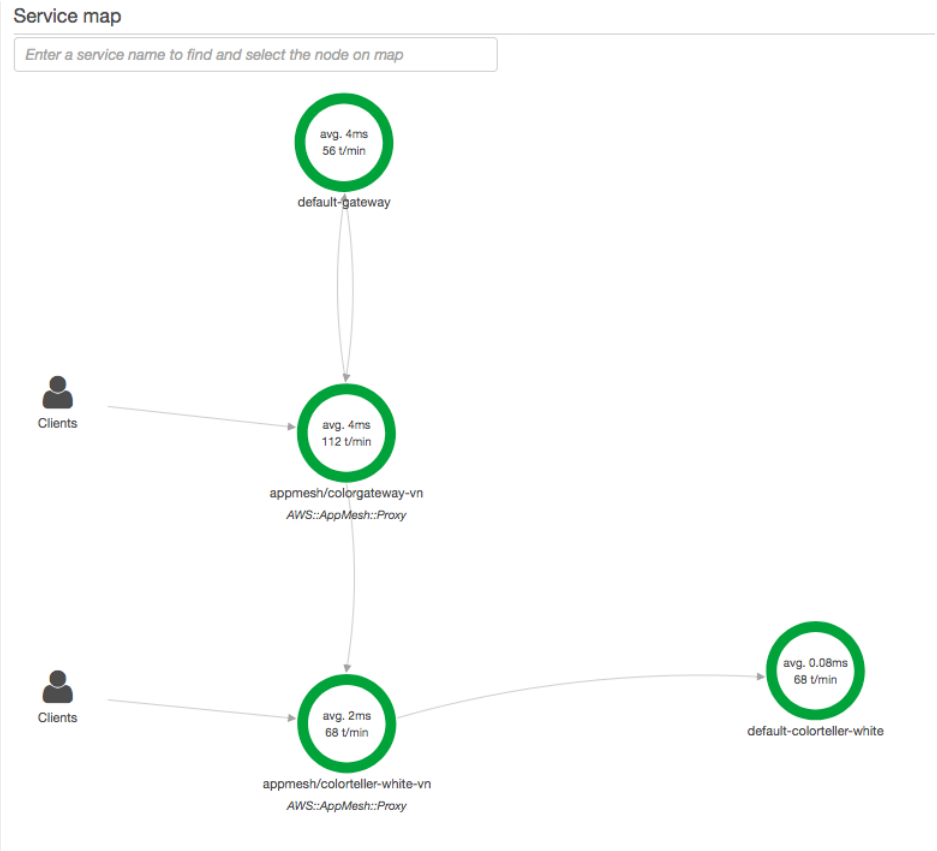
For more information, see Trace API Gateway API Execution with AWS X-Ray in the Amazon API Gateway Developer Guide.

# Amazon EC2 and AWS App Mesh

AWS X-Ray integrates with AWS App Mesh to manage Envoy proxies for microservices. App Mesh provides a version of Envoy that you can configure to send trace data to the X-Ray daemon running in a container of the same task or pod. X-Ray supports tracing with the following App Mesh compatible services:

- Amazon Elastic Container Service (Amazon ECS)
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Amazon Elastic Compute Cloud (Amazon EC2)

Use the following instructions to learn how to enable X-Ray tracing through App Mesh.




To configure the Envoy proxy to send data to X-Ray, set the `ENABLE_ENVOY_XRAY_TRACING` environment variable in its container definition.

### Example Envoy container definition for Amazon ECS

```
{
      "name": "envoy",
      "image": "840364872350.dkr.ecr.us-west-2.amazonaws.com/aws-appmesh-envoy:v1.12.3.0-
prod",
      "essential": true,
      "environment": [
        {
          "name": "APPMESH_VIRTUAL_NODE_NAME",
          "value": "mesh/myMesh/virtualNode/myNode"
        },
        {
          "name": "ENABLE_ENVOY_XRAY_TRACING",
          "value": "1"
        }
      ],
      "healthCheck": {
        "command": [
          "CMD-SHELL",
          "curl -s http://localhost:9901/server_info | cut -d' ' -f3 | grep -q live"
        ],
        "startPeriod": 10,
        "interval": 5,
        "timeout": 2,
        "retries": 3
      }
```

**Note**
To learn more about available Envoy region addresses, see Envoy image in the AWS App Mesh User Guide.

For details on running the X-Ray daemon in a container, see Running the X-Ray daemon on Amazon ECS (p. 149). For a sample application that includes a service mesh, microservice, Envoy proxy, and X-Ray daemon, deploy the `colorapp` sample in the App Mesh Examples GitHub repository.

### Learn More

- Getting Started with AWS App Mesh
- Getting Started with AWS App Mesh and Amazon ECS

# AWS AppSync and AWS X-Ray

You can enable and trace requests for AWS AppSync. For more information, see Tracing with AWS X-Ray for instructions.

When X-Ray tracing is enabled for an AWS AppSync API, an AWS Identity and Access Management service-linked role is automatically created in your account with the appropriate permissions. This allows AWS AppSync to send traces to X-Ray in a secure way.

# Logging X-Ray API calls with AWS CloudTrail

AWS X-Ray integrates with AWS CloudTrail to record API actions made by a user, a role, or an AWS service in X-Ray. You can use CloudTrail to monitor X-Ray API requests in real time and store logs in Amazon S3, Amazon CloudWatch Logs, and Amazon CloudWatch Events. X-Ray supports logging the following actions as events in CloudTrail log files:

### Supported API Actions

- `PutEncryptionConfig`
- `GetEncryptionConfig`
- `CreateGroup`
- `UpdateGroup`
- `DeleteGroup`
- `GetGroup`
- `GetGroups`

### To create a trail

1. Open the **Trails** page of the CloudTrail console.
2. Choose **Create trail**.
3. Enter a trail name, and then choose the types of event to record.
   - **Management events** – Record API actions that create, read, update, or delete AWS resources. Records calls to all supported API actions for all AWS services.
   - **Data events** – Record API actions that target specific resources, like Amazon S3 object reads or AWS Lambda function invocations. You choose which buckets and functions to monitor.

4. Choose an Amazon S3 bucket and encryption settings.
5. Choose **Create**.

CloudTrail records API calls of the types you chose to log files in Amazon S3. A CloudTrail log is an unordered array of events in JSON format. For each call to a supported API action, CloudTrail records information about the request and the entity that made it. Log events include the action name, parameters, the response from X-Ray, and details about the requester.

### Example X-Ray GetEncryptionConfig log entry

```
{
    "eventVersion"=>"1.05",
    "userIdentity"=>{
        "type"=>"AssumedRole",
        "principalId"=>"AROAJVHBZWD3DN6CI2MHM:MyName",
        "arn"=>"arn:aws:sts::123456789012:assumed-role/MyRole/MyName",
        "accountId"=>"123456789012",
        "accessKeyId"=>"AKIAIOSFODNN7EXAMPLE",
        "sessionContext"=>{
            "attributes"=>{
                "mfaAuthenticated"=>"false",
                "creationDate"=>"2018-9-01T00:24:36Z"
            },
            "sessionIssuer"=>{
                "type"=>"Role",
                "principalId"=>"AROAJVHBZWD3DN6CI2MHM",
                "arn"=>"arn:aws:iam::123456789012:role/MyRole",
                "accountId"=>"123456789012",
                "userName"=>"MyRole"
            }
        }
    },
    "eventTime"=>"2018-9-01T00:24:36Z",
    "eventSource"=>"xray.amazonaws.com",
    "eventName"=>"GetEncryptionConfig",
    "awsRegion"=>"us-east-2",
    "sourceIPAddress"=>"33.255.33.255",
    "userAgent"=>"aws-sdk-ruby2/2.11.19 ruby/2.3.1 x86_64-linux",
    "requestParameters"=>nil,
    "responseElements"=>nil,
    "requestID"=>"3fda699a-32e7-4c20-37af-edc2be5acbdb",
    "eventID"=>"039c3d45-6baa-11e3-2f3e-e5a036343c9f",
    "eventType"=>"AwsApiCall",
    "recipientAccountId"=>"123456789012"
}
```

The userIdentity element contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or 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.

To be notified when a new log file is available, configure CloudTrail to publish Amazon SNS notifications. For more information, see Configuring Amazon SNS Notifications for CloudTrail.

You can also aggregate X-Ray log files from multiple AWS Regions and multiple AWS accounts into a single Amazon S3 bucket. For more information, see Receiving CloudTrail Log Files from Multiple Regions and Receiving CloudTrail Log Files from Multiple Accounts.

# Monitoring endpoints and APIs with CloudWatch

AWS X-Ray integrates with Amazon CloudWatch to support CloudWatch ServiceLens and CloudWatch Synthetics in monitoring the health of your applications. By correlating metrics, logs, and traces, ServiceLens provides an end-to-end view of your services to help you quickly pinpoint performance bottlenecks and identify impacted users. To learn more about ServiceLens, see Using ServiceLens to Monitor the Health of Your Applications.

ServiceLens integrates with CloudWatch Synthetics, a fully managed service that enables you to monitor your endpoints and APIs from the outside in. Synthetics uses modular, lightweight canaries that run 24 hours per day, once per minute. Canaries are configurable scripts that follow the same routes and perform the same actions as a customer. This enables the outside-in view of your customers' experiences, and your service's availability from their point of view.

You can customize canaries to check for availability, latency, transactions, broken or dead links, step-by-step task completions, page load errors, load latency for UI assets, complex wizard flows, or other workflows in your application.

To get started with Synthetics, enable X-Ray for your APIs, endpoints, and web apps, for example your APIs running on API Gateway (p. 153). Then create a canary and observe the Synthetics node in your service graph. To learn more about setting up Synthetics tests, see Using Synthetics to Create and Manage Canaries.

**Topics**



## Debugging CloudWatch synthetics canaries using X-Ray

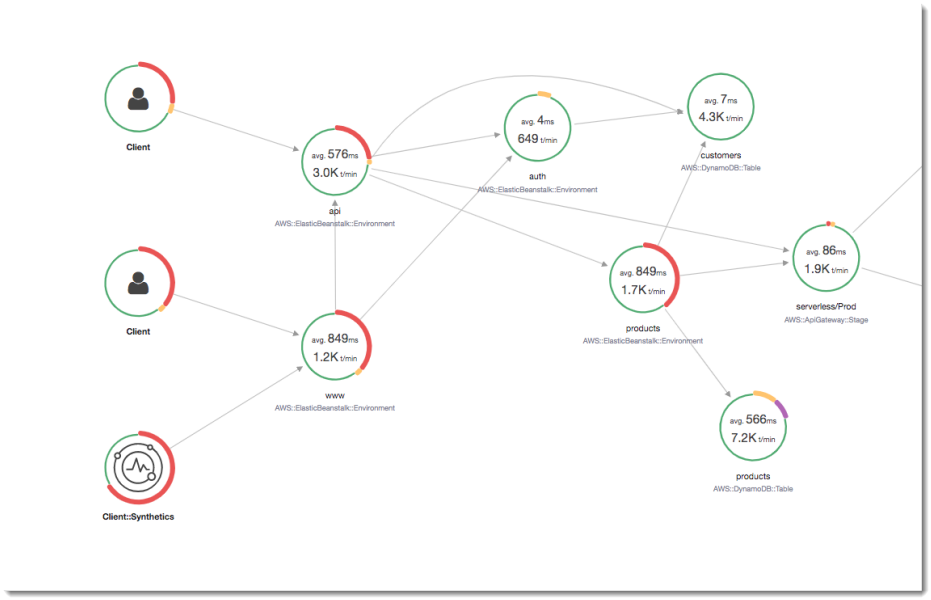
CloudWatch Synthetics is a fully managed service that enables you to monitor your endpoints and APIs using scripted canaries that run 24 hours per day, once per minute.

You can customize canary scripts to check for changes in:

- Availability
- Latency
- Transactions
- Broken or dead links
- Step-by-step task completions
- Page load errors
- Load Latencies for UI assets
- Complex wizard flows
- Checkout flows in your application

Canaries follow the same routes and perform the same actions and behaviors as your customers, and continually verify the customer experience.

To learn more about setting up Synthetics tests, see Using Synthetics to Create and Manage Canaries.




The following examples show common use cases for debugging issues that your Synthetics canaries raise. Each example demonstrates a key strategy for debugging using either the service map or the X-Ray Analytics console.

For more information about how to read and interact with the service map, see Viewing the Service Map.

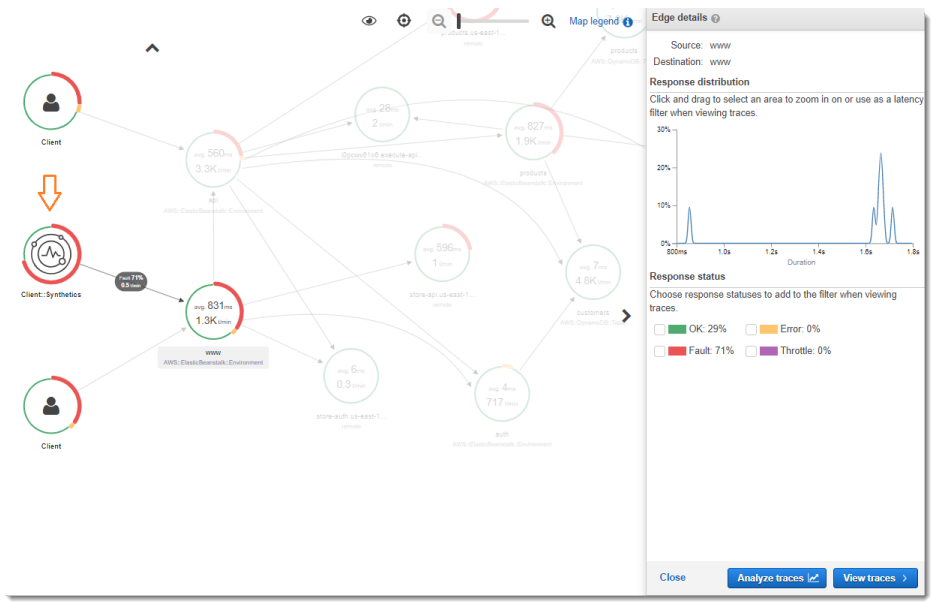
For more information about how to read and interact with the X-Ray Analytics console, see Interacting with the AWS X-Ray Analytics Console.

**Topics**



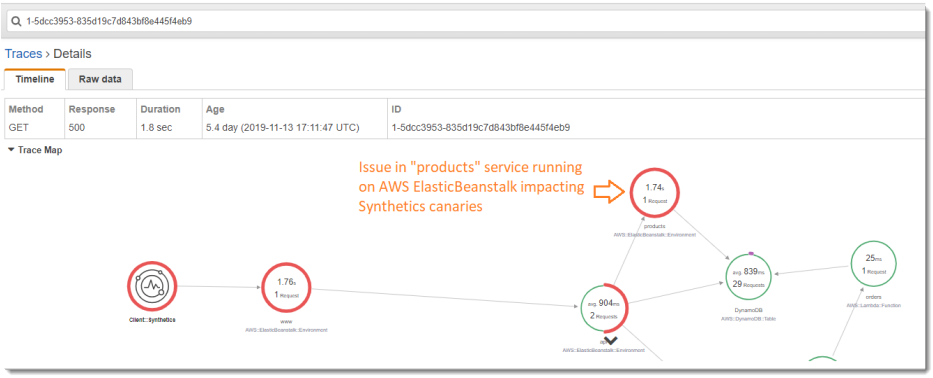
## View canaries with increased error reporting in the service map

To see which canaries have an increase in errors, faults, throttling rates, or slow response times within your X-Ray service map, you can highlight Synthetics canary client nodes using the `Client::Synthetic` filter (p. 59). Clicking a Synthetics canary node displays the response time distribution of the entire request.




## Use trace maps for individual traces to view each request in detail

To determine which service results in the most latency or is causing an error, invoke the trace map by selecting the trace in the service map. Individual trace maps display the end-to-end path of a single request. Use this to understand the services invoked, and visualize the upstream and downstream services.



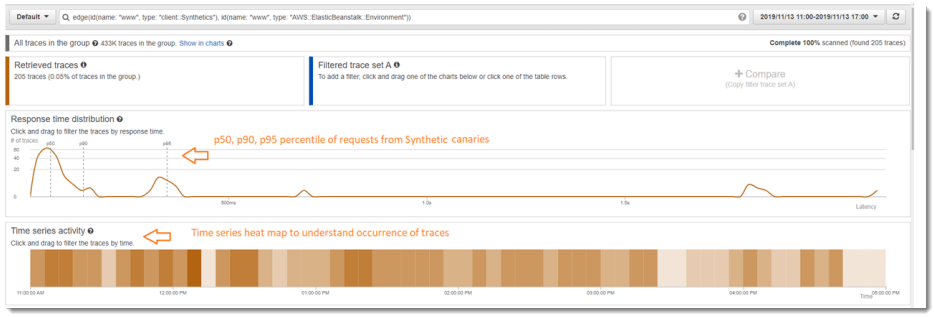

## Determine the root cause of ongoing failures in upstream and downstream services

Once you receive a CloudWatch alarm for failures in a Synthetics canary, use the statistical modeling on trace data in X-Ray to determine the probable root cause of the issue within the X-Ray Analytics console. In the Analytics console, the **Response Time Root Cause** table shows recorded entity paths. X-Ray determines which path in your trace is the most likely cause for the response time. The format indicates a hierarchy of entities that are encountered, ending in a response time root cause.

The following example shows that the Synthetics test for API “XXX” running on API Gateway is failing due to a throughput capacity exception from the Amazon DynamoDB table.

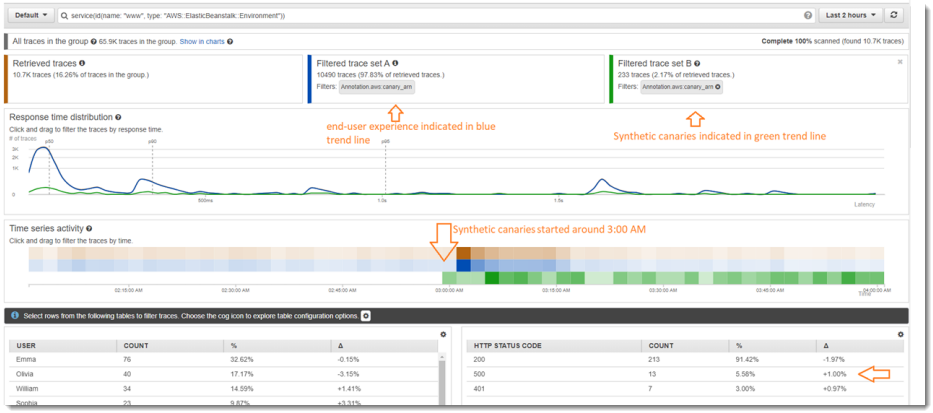

# Identify performance bottlenecks and trends

You can view trends in the performance of your endpoint over time using continuous traffic from your Synthetics canaries to populate a trace map over a period of time.



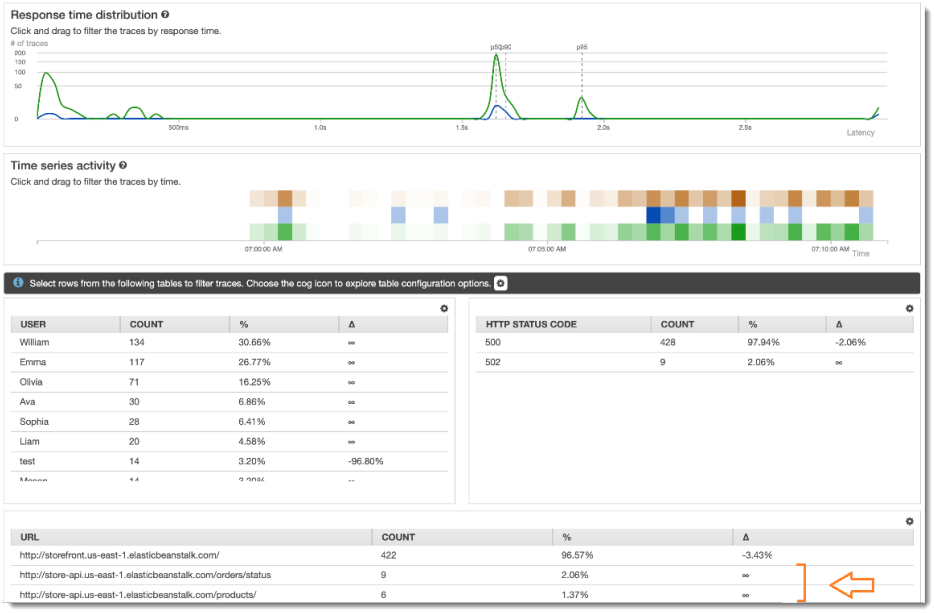
## Compare latency and error or fault rates before and after changes

Pinpoint the time a change occured to correlate that change to an increase in issues caught by your canaries. Use the X-Ray Analytics console to define the before and after time ranges as different trace sets, creating a visual differentiation in the response time distribution.




## Determine the required canary coverage for all APIs and URLs

Use X-Ray Analytics to compare the experience of canaries with the users. The UI below shows a blue trend line for canaries and a green line for the users. You can also identify that two out of the three URLs don't have canary tests.

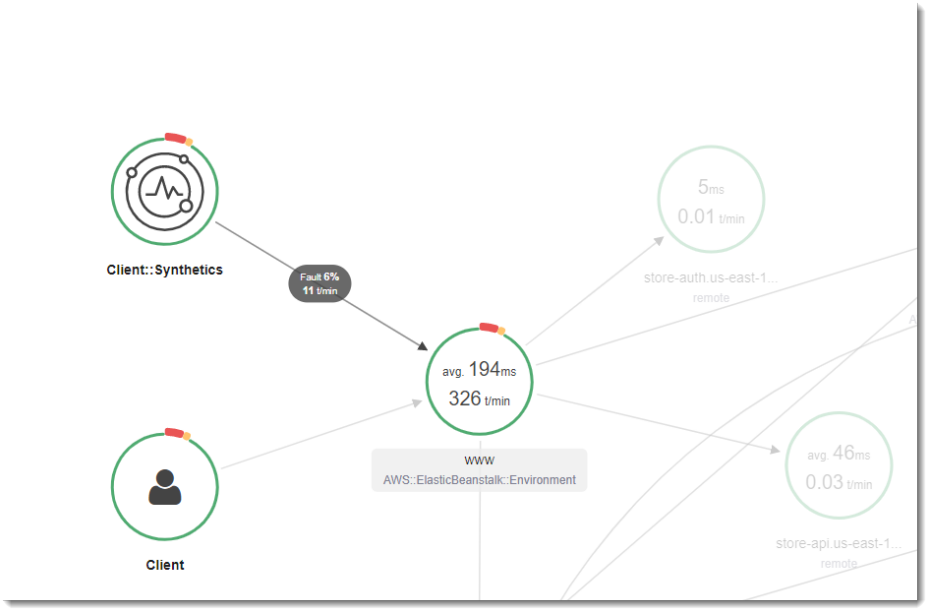



## Use groups to focus on synthetics tests

To focus on a certain set of workflows, such as a Synthetics tests for application "www" running on AWS Elastic Beanstalk. You can create an X-Ray group using a filter expression.

**Example Group filter expression**

```
"edge(id(name: "www", type: "client::Synthetics"), id(name: "www", type:
 "AWS::ElasticBeanstalk::Environment"))"
```




# Tracking X-Ray encryption configuration changes with AWS Config

AWS X-Ray integrates with AWS Config to record configuration changes made to your X-Ray encryption resources. You can use AWS Config to inventory X-Ray encryption resources, audit the X-Ray configuration history, and send notifications based on resource changes.

AWS Config supports logging the following X-Ray encryption resource changes as events:

- **Configuration changes** – Changing or adding an encryption key, or reverting to the default X-Ray encryption setting.

Use the following instructions to learn how to create a basic connection between X-Ray and AWS Config.

## Creating a Lambda function trigger

You must have the ARN of a custom AWS Lambda function before you can generate a custom AWS Config rule. Follow these instructions to create a basic function with Node.js that returns a compliant or non-compliant value back to AWS Config based on the state of the `XrayEncryptionConfig` resource.

**To create a Lambda function with an AWS::XrayEncryptionConfig change trigger**

1. Open the Lambda console. Choose **Create function**.
2. Choose **Blueprints**, and then filter the blueprints library for the **config-rule-change-triggered** blueprint. Either click the link in the blueprint's name or choose **Configure** to continue.
3. Define the following fields to configure the blueprint:

- For **Name**, type a name.
- For **Role**, choose **Create new role from template(s)**.
- For **Role name**, type a name.
- For **Policy templates**, choose **AWS Config Rules permissions**.

4. Choose **Create function** to create and display your function in the AWS Lambda console.
5. Edit your function code to replace `AWS::EC2::Instance` with `AWS::XrayEncryptionConfig`. You can also update the description field to reflect this change.

   **Default Code**

```
   if (configurationItem.resourceType !== 'AWS::EC2::Instance') {
       return 'NOT_APPLICABLE';
   } else if (ruleParameters.desiredInstanceType ===
configurationItem.configuration.instanceType) {
       return 'COMPLIANT';
   }
       return 'NON_COMPLIANT';
```

   **Updated Code**

```
   if (configurationItem.resourceType !== 'AWS::XRay::EncryptionConfig') {
       return 'NOT_APPLICABLE';
   } else if (ruleParameters.desiredInstanceType ===
configurationItem.configuration.instanceType) {
       return 'COMPLIANT';
   }
       return 'NON_COMPLIANT';
```

6. Add the following to your execution role in IAM for access to X-Ray. These permissions allow read-only access to your X-Ray resources. Failure to provide access to the appropriate resources will result in an out of scope message from AWS Config when it evaluates the Lambda function associated with the rule.

```
   {
       "Sid": "Stmt1529350291539",
       "Action": [
           "xray:GetEncryptionConfig"
       ],
       "Effect": "Allow",
       "Resource": "*"
    }
```

# Creating a custom AWS Config rule for x-ray

When the Lambda function is created, note the function's ARN, and go to the AWS Config console to create your custom rule.

### To create an AWS Config rule for X-Ray

1. Open the **Rules** page of the AWS Config console.
2. Choose **Add rule**, and then choose **Add custom rule**.
3. In **AWS Lambda Function ARN**, insert the ARN associated with the Lambda function you want to use.
4. Choose the type of trigger to set:

- **Configuration changes** – AWS Config triggers the evaluation when any resource that matches the rule's scope changes in configuration. The evaluation runs after AWS Config sends a configuration item change notification.
- **Periodic** – AWS Config runs evaluations for the rule at a frequency that you choose (for example, every 24 hours).

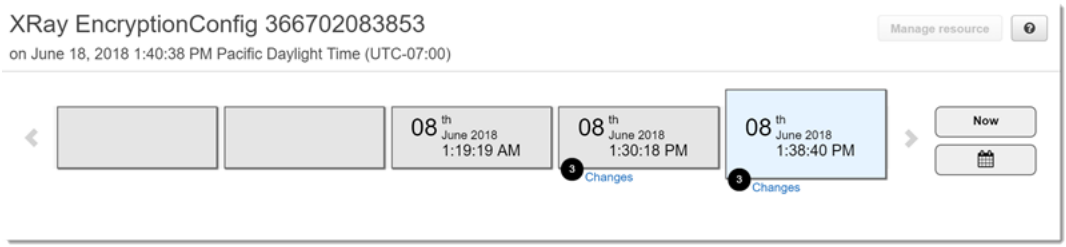
5. For **Resource type**, choose **EncryptionConfig** in the X-Ray section.
6. Choose **Save**.

The AWS Config console begins to evaluate the rule's compliance immediately. The evaluation can take several minutes to complete.

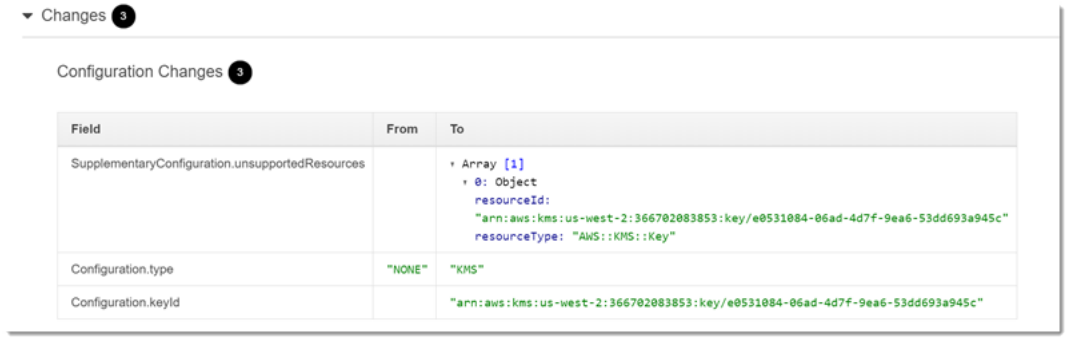
Now that this rule is compliant, AWS Config can begin to compile an audit history. AWS Config records resource changes in the form of a timeline. For each change in the timeline of events, AWS Config generates a table in a from/to format to show what changed in the JSON representation of the encryption key. The two field changes associated with EncryptionConfig are `Configuration.type` and `Configuration.keyID`.

# Example results

Following is an example of an AWS Config timeline showing changes made at specific dates and times.




Following is an example of an AWS Config change entry. The from/to format illustrates what changed. This example shows that the default X-Ray encryption settings were changed to a defined encryption key.




# Amazon SNS notifications

To be notified of configuration changes, set AWS Config to publish Amazon SNS notifications. For more information, see Monitoring AWS Config Resource Changes by Email.

# Amazon Elastic Compute Cloud and AWS X-Ray

You can install and run the X-Ray daemon on an Amazon EC2 instance with a user data script. See Running the X-Ray daemon on Amazon EC2 (p. 148) for instructions.

Use an instance profile to grant the daemon permission to upload trace data to X-Ray. For more information, see Giving the daemon permission to send data to X-Ray (p. 139).

# AWS Elastic Beanstalk and AWS X-Ray

AWS Elastic Beanstalk platforms include the X-Ray daemon. You can run the daemon (p. 145) by setting an option in the Elastic Beanstalk console or with a configuration file.

On the Java SE platform, you can use a Buildfile file to build your application with Maven or Gradle on-instance. The X-Ray SDK for Java and AWS SDK for Java are available from Maven, so you can deploy only your application code and build on-instance to avoid bundling and uploading all of your dependencies.

You can use Elastic Beanstalk environment properties to configure the X-Ray SDK. The method that Elastic Beanstalk uses to pass environment properties to your application varies by platform. Use the X-Ray SDK's environment variables or system properties depending on your platform.

- **Node.js platform** – Use environment variables (p. 219)
- **Java SE platform** – Use environment variables (p. 195)
- **Tomcat platform** – Use system properties (p. 196)

For more information, see Configuring AWS X-Ray Debugging in the AWS Elastic Beanstalk Developer Guide.

# Elastic Load Balancing and AWS X-Ray

Elastic Load Balancing application load balancers add a trace ID to incoming HTTP requests in a header named `X-Amzn-Trace-Id`.

```
X-Amzn-Trace-Id: Root=1-5759e988-bd862e3fe1be46a994272793
```

### Trace ID Format

A `trace_id` consists of three numbers separated by hyphens. For example, `1-58406520-a006649127e371903a2de979`. This includes:

- The version number, that is, `1`.
- The time of the original request, in Unix epoch time, in **8 hexadecimal digits**.

  For example, 10:00AM December 1st, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal digits.
- A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.

Load balancers do not send data to X-Ray, and do not appear as a node on your service map.

For more information, see Request Tracing for Your Application Load Balancer in the Elastic Load Balancing Developer Guide.

# AWS Lambda and AWS X-Ray

You can use AWS X-Ray to trace your AWS Lambda functions. Lambda runs the X-Ray daemon (p. 137) and records a segment with details about the function invocation and execution. For further instrumentation, you can bundle the X-Ray SDK with your function to record outgoing calls and add annotations and metadata.

If your Lambda function is called by another instrumented service, Lambda traces requests that have already been sampled without any additional configuration. The upstream service can be an instrumented web application or another Lambda function. Your service can invoke the function directly with an instrumented AWS SDK client, or by calling an API Gateway API with an instrumented HTTP client.

If your Lambda function runs on a schedule, or is invoked by a service that is not instrumented, you can configure Lambda to sample and record invocations with active tracing.

**To configure X-Ray integration on an AWS Lambda function**

1. Open the AWS Lambda console.
2. Choose your function.
3. Choose **Configuration**.
4. Under **AWS X-Ray**, enable **Active tracing**.

On runtimes with a corresponding X-Ray SDK, Lambda also runs the X-Ray daemon.

**X-Ray SDKs on Lambda**

- **X-Ray SDK for Go** – Go 1.7 and newer runtimes
- **X-Ray SDK for Java** – Java 8 runtime
- **X-Ray SDK for Node.js** – Node.js 4.3 and newer runtimes
- **X-Ray SDK for Python** – Python 2.7, Python 3.6, and newer runtimes
- **X-Ray SDK for .NET** – .NET Core 2.0 and newer runtimes

To use the X-Ray SDK on Lambda, bundle it with your function code each time you create a new version. You can instrument your Lambda functions with the same methods that you use to instrument applications running on other services. The primary difference is that you don't use the SDK to instrument incoming requests, make sampling decisions, and create segments.

The other difference between instrumenting Lambda functions and web applications is that the segment that Lambda creates and sends to X-Ray cannot be modified by your function code. You can create subsegments and record annotations and metadata on them, but you can't add annotations and metadata to the parent segment.

For more information, see Using AWS X-Ray in the *AWS Lambda Developer Guide*.

# Amazon SNS and AWS X-Ray

AWS X-Ray integrates with Amazon Simple Notification Service (Amazon SNS) to trace messages that are passed through Amazon SNS. If an Amazon SNS publisher traces its client with the X-Ray SDK,

subscribers can retrieve the tracing header (p. 26) and continue to propagate the original trace from the publisher with the same trace ID. This continuity enables users to trace, analyze, and debug throughout downstream services.

Amazon SNS trace context propagation currently supports the following subscribers:

- **HTTP/HTTPS** – For HTTP/HTTPS subscribers, you can use the X-Ray SDK to trace the incoming message request. For more information and examples in Java, see Tracing incoming requests with the X-Ray SDK for Java (p. 196).
- **AWS Lambda** – For Lambda subscribers with active tracing enabled, Lambda records a segment with details about the function invocation, and sends it to the publisher's trace. For more information, see AWS Lambda and AWS X-Ray (p. 167).

Use the following instructions to learn how to create a basic context between X-Ray and Amazon SNS using a Lambda subscriber. You will create two Lambda functions and an Amazon SNS topic. Then, in the X-Ray console, you can view the trace ID propagated throughout their interactions.

# Requirements

- Node.js 8 with `npm`
- The Bash shell. For Linux and macOS, this is included by default. In Windows 10, you can install the Windows Subsystem for Linux to get a Windows-integrated version of Ubuntu and Bash.
- The AWS CLI

# Creating a Lambda subscriber function

In the following steps, the sample Lambda function *`MessageSubscriber`* is implemented in Node.js and is subscribed as an endpoint to an Amazon SNS topic. The *`MessageSubscriber`* function creates a `custom process_message` subsegment with the `AWSXRay.captureFunc` command. The function writes a simple message as an annotation to the subsegment.

### To create the subscriber package

1. Create a file folder and name it to indicate it's the subscriber function (for example, *`sns-xray-subscriber`*).
2. Create two files: `index.js` and `package.json`.
3. Paste the following code into `index.js`.

```
var AWSXRay = require('aws-xray-sdk');

exports.handler = function(event, context, callback) {
    AWSXRay.captureFunc('process_message', function(subsegment) {
        var message = event.Records[0].Sns.Message;
        subsegment.addAnnotation('message_content', message);
        subsegment.close();
    });

    callback(null, "Message received.");
};
```

4. Paste the following code into `package.json`.

```
{
  "name": "sns-xray-subscriber",
```

```
  "version": "1.0.0",
  "description": "A demo service to test SNS X-Ray trace header propagation",
  "dependencies": {
    "aws-xray-sdk": "^2.2.0"
  }
}
```

5. Run the following script within the *sns-xray-subscriber* folder. It creates a `package-lock.json` file and a `node_modules` folder, which handle all dependencies.

```
npm install --production
```

6. Compress the *sns-xray-subscriber* folder into a .zip file.

**To create a subscriber function and enable X-Ray**

1. Open the Lambda console, and then choose **Create a function**.
2. Choose **Author from scratch**:
   - For **Function name**, provide a name (for example, *MessageSubscriber*).
   - For **Runtime**, use **Node.js 10.x**.
3. Choose **Create function** to create and display your function in the Lambda console.
4. In **Function code**, under **Code entry type**, choose **Upload a .zip file**.
5. Choose the subscriber package you created. To upload, choose **Save** in the upper right of the console.
6. In **Debugging and error handling**, select the **Enable AWS X-Ray** box.
7. Choose **Save**.

# Creating an Amazon SNS topic

When Amazon SNS receives requests, it propagates the trace header to its endpoint subscriber. In the following steps, you create a topic and then set the endpoint as the Lambda function you created earlier.

**To create an Amazon SNS topic and subscribe a Lambda function**

1. Open the SNS console.
2. Choose **Topics**, and then choose **Create topic**. For **Name**, provide a name.
3. In **Subscriptions**, choose **Create subscription**.
4. Record the topic ARN (for example, `arn:aws:sns:{region}:{account id}:{topic name}`).
5. Choose **Create subscription**:
   - For **Protocol**, choose **AWS Lambda**.
   - For **Endpoint**, choose the ARN of the receiver Lambda function you created from the list of available Lambda functions.
6. Choose **Create subscription**.

# Creating a Lambda publisher function

In the following steps, the sample Lambda function *MessagePublisher* is implemented in Node.js. The function sends a message to the Amazon SNS topic you created earlier. The function uses the AWS SDK

for JavaScript to send notifications from Amazon SNS, and the X-Ray SDK for Node.js to instrument the AWS SDK client.

### To create the publisher package

1. Create a file folder and name it to indicate it's the publisher function (for example, *sns-xray-publisher*).
2. Create two files: `index.js` and `package.json`.
3. Paste the following code into `index.js`.

```
var AWSXRay = require('aws-xray-sdk');
var AWS = AWSXRay.captureAWS(require('aws-sdk'));

exports.handler = function(event, context, callback) {
    var sns = new AWS.SNS();

    sns.publish({
  // You can replace the following line with your custom message.
        Message: process.env.MESSAGE || "Testing X-Ray trace header propagation",
        TopicArn: process.env.TOPIC_ARN
    }, function(err, data) {
        if (err) {
            console.log(err.stack);
            callback(err);
        } else {
            callback(null, "Message sent.");
        }
    });
};
```

4. Paste the following code into `package.json`.

```
{
  "name": "sns-xray-publisher",
  "version": "1.0.0",
  "description": "A demo service to test SNS X-Ray trace header propagation",
  "dependencies": {
    "aws-xray-sdk": "^2.2.0"
  }
}
```

5. Run the following script within the *sns-xray-publisher* folder. It creates a `package-lock.json` file and a `node_modules` folder, which handle all dependencies.

```
npm install --production
```

6. Compress the *sns-xray-publisher* folder into a .zip file.

### To create a publisher function and enable X-Ray

1. Open the Lambda console, and then choose **Create function**.
2. Choose **Author from scratch**:
   - For **Name**, provide a name (for example, *MessagePublisher*).
   - For **Runtime**, use **Node.js 10.x**.
3. In **Permissions**, expand **Choose or create an execution role**:
   - For **Execution role**, choose **Create a new role from AWS policy templates**.
   - For **Role name**, provide a name.

   - For **Policy templates**, choose **Amazon SNS publish policy**.
4. Choose **Create function** to create and display your function in the Lambda console.
5. In **Function code**, under **Code entry type**, choose **Upload a .zip file**.
6. Choose the publisher package that you created. To upload, choose **Save** in the upper right of the console.
7. In **Environment variables**, add a variable:
   - For **Key**, use the key name `TOPIC_ARN`, which is defined in the publisher function.
   - For **Value**, use the Amazon SNS topic ARN you recorded previously.
8. Optionally, add another variable:
   - For **Key**, provide a key name (for example, *`MESSAGE`*).
   - For **Value**, enter any custom message.
9. In **Debugging and error handling**, select the **Enable AWS X-Ray** box.
10. Choose **Save**.

## Testing and validating context propagation

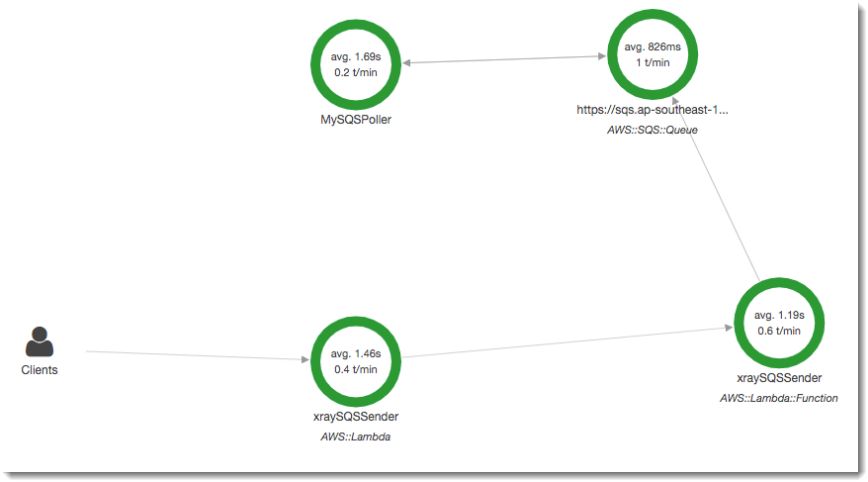
Both publisher and subscriber functions enable Lambda active tracing when sending traces. The publisher function uses the X-Ray SDK to capture the `publish` SNS API call. Then, Amazon SNS propagates the trace header to the subscriber. Finally, the subscriber picks up the trace header and continues the trace. Follow the trace ID in the following steps.

**To create a publisher function and enable X-Ray**

1. Open the Lambda console, and then choose the publisher function that you created previously.
2. Choose **Test**.
3. Choose **Create a new test event**:
   - For **Event template**, choose the **Hello World** template.
   - For **Event name**, provide a name.
4. Choose **Create**.
5. Choose **Test** again.
6. To verify, open the X-Ray console. Wait at least 10 seconds for the trace to appear.
7. When the service map is generated, validate that your two Lambda functions and the Amazon SNS topic appear.
8. Choose the *`MessageSubscriber`* segment, and then choose **View traces**.
9. Choose the trace from the list to reach the **Details** page.
10. Choose the **process_message** subsegment.
11. Choose the **Annotations** tab to see the `message_content` key with the message value from the sender.

# Amazon SQS and AWS X-Ray

AWS X-Ray integrates with Amazon Simple Queue Service (Amazon SQS) to trace messages that are passed through an Amazon SQS queue. If a service traces requests by using the X-Ray SDK, Amazon SQS can send the tracing header and continue to propagate the original trace from the sender to the consumer with a consistent trace ID. Trace continuity enables users to track, analyze, and debug throughout downstream services.




Amazon SQS supports the following tracing header instrumentation:

- **Default HTTP Header** – The X-Ray SDK automatically populates the trace header as an HTTP header when you call Amazon SQS through the AWS SDK. The default trace header is carried by `X-Amzn-Trace-Id` and corresponds to all messages included in a `SendMessage` or `SendMessageBatch` request. To learn more about the default HTTP header, see Tracing header (p. 26).
- **`AWSTraceHeader` System Attribute** – The `AWSTraceHeader` is a message system attribute reserved by Amazon SQS to carry the X-Ray trace header with messages in the queue. `AWSTraceHeader` is available for use even when auto-instrumentation through the X-Ray SDK is not, for example when building a tracing SDK for a new language. When both header instrumentations are set, the message system attribute overrides the HTTP trace header.

When running on Amazon EC2, Amazon SQS supports processing one message at a time. This applies when running on an on-premises host, and when using container services, such as AWS Fargate, Amazon ECS, or AWS App Mesh.

The trace header is excluded from both Amazon SQS message size and message attribute quoatas. Enabling X-Ray tracing will not exceed your Amazon SQS quotas. To learn more about AWS quotas, see Amazon SQS Quotas.

# Send the HTTP trace header

Sender components in Amazon SQS can send the trace header automatically through the `SendMessageBatch` or `SendMessage` call. When AWS SDK clients are instrumented, they can be automatically tracked through all languages supported through the X-Ray SDK. Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the service map in the X-Ray console.

To learn how to trace AWS SDK calls with your preferred language, see the following topics in the supported SDKs:

- Go – Tracing AWS SDK calls with the X-Ray SDK for Go (p. 181)
- Java – Tracing AWS SDK calls with the X-Ray SDK for Java (p. 199)
- Node.js – Tracing AWS SDK calls with the X-Ray SDK for Node.js (p. 222)
- Python – Tracing AWS SDK calls with the X-Ray SDK for Python (p. 242)
- Ruby – Tracing AWS SDK calls with the X-Ray SDK for Ruby (p. 261)
- .NET – Tracing AWS SDK calls with the X-Ray SDK for .NET (p. 274)

# Retrieve the trace header and recover trace context

To continue context propagation with Amazon SQS, you must manually instrument the handoff to the receiver component.

There are three main steps to recovering the trace context:

- Receive the message from the queue for the `AWSTraceHeader` attribute by calling the `ReceiveMessage` API.
- Retrieve the trace header from the attribute.
- Recover the trace ID from the header. Optionally, add more metrics to the segment.

The following is an example implementation written with the X-Ray SDK for Java.

**Example : Retrieve the trace header and recover trace context**

```
// Receive the message from the queue, specifying the "AWSTraceHeader"
ReceiveMessageRequest receiveMessageRequest = new ReceiveMessageRequest()
        .withQueueUrl(QUEUE_URL)
        .withAttributeNames("AWSTraceHeader");
List<Message> messages = sqs.receiveMessage(receiveMessageRequest).getMessages();

if (!messages.isEmpty()) {
    Message message = messages.get(0);

    // Retrieve the trace header from the AWSTraceHeader message system attribute
    String traceHeaderStr = message.getAttributes().get("AWSTraceHeader");
    if (traceHeaderStr != null) {
        TraceHeader traceHeader = TraceHeader.fromString(traceHeaderStr);

        // Recover the trace context from the trace header
        Segment segment = AWSXRay.getCurrentSegment();
        segment.setTraceId(traceHeader.getRootTraceId());
        segment.setParentId(traceHeader.getParentId());

 segment.setSampled(traceHeader.getSampled().equals(TraceHeader.SampleDecision.SAMPLED));
    }
}
```

# AWS X-Ray SDK for Go

The X-Ray SDK for Go is a set of libraries for Go applications that provide classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an SQL database connector. You can also create segments manually and add debug information in annotations and metadata.

Download the SDK from its GitHub repository with `go get`:

```
$ go get -u github.com/aws/aws-xray-sdk-go/...
```

For web applications, start by using the `xray.Handler` function (p. 180) to trace incoming requests. The message handler creates a segment (p. 21) for each traced request, and completes the segment when the response is sent. While the segment is open you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header (p. 26) and traces sampled requests automatically. For other functions, you can configure Lambda (p. 167) to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

> **Note**
> On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray (p. 167) for more information.

Next, wrap your client with a call to the `AWS` function (p. 181). This step ensures that X-Ray instruments calls to any client methods. You can also instrument calls to SQL databases (p. 183).

Once you get going with the SDK, customize its behavior by configuring the recorder and middleware (p. 175). You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata (p. 184). Annotations are simple key-value pairs that are indexed for use with filter expressions (p. 59), so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

> **Annotations and Metadata**
> Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments (p. 183). You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

# Requirements

The X-Ray SDK for Go requires Go 1.7 or later.

The SDK depends on the following libraries at compile and runtime:

- AWS SDK for Go version 1.10.0 or newer

These dependencies are declared in the SDK's `README.md` file.

# Reference documentation

Once you have downloaded the SDK, build and host the documentation locally to view it in a web browser.

**To view the reference documentation**

1. Navigating to the `$GOPATH/src/github.com/aws/aws-xray-sdk-go` (Linux or Mac) directory or the `%GOPATH%\src\github.com\aws\aws-xray-sdk-go` (Windows) folder
2. Run the `godoc` command.

   ```
   $ godoc -http=:6060
   ```

3. Opening a browser at `http://localhost:6060/pkg/github.com/aws/aws-xray-sdk-go/`.

# Configuring the X-Ray SDK for Go

You can specify the configuration for for X-Ray SDK for Go through environment variables, by calling `Configure` with a `Config` object, or by assuming default values. Environment variables take precedence over `Config` values, which take precedence over any default value.

**Sections**



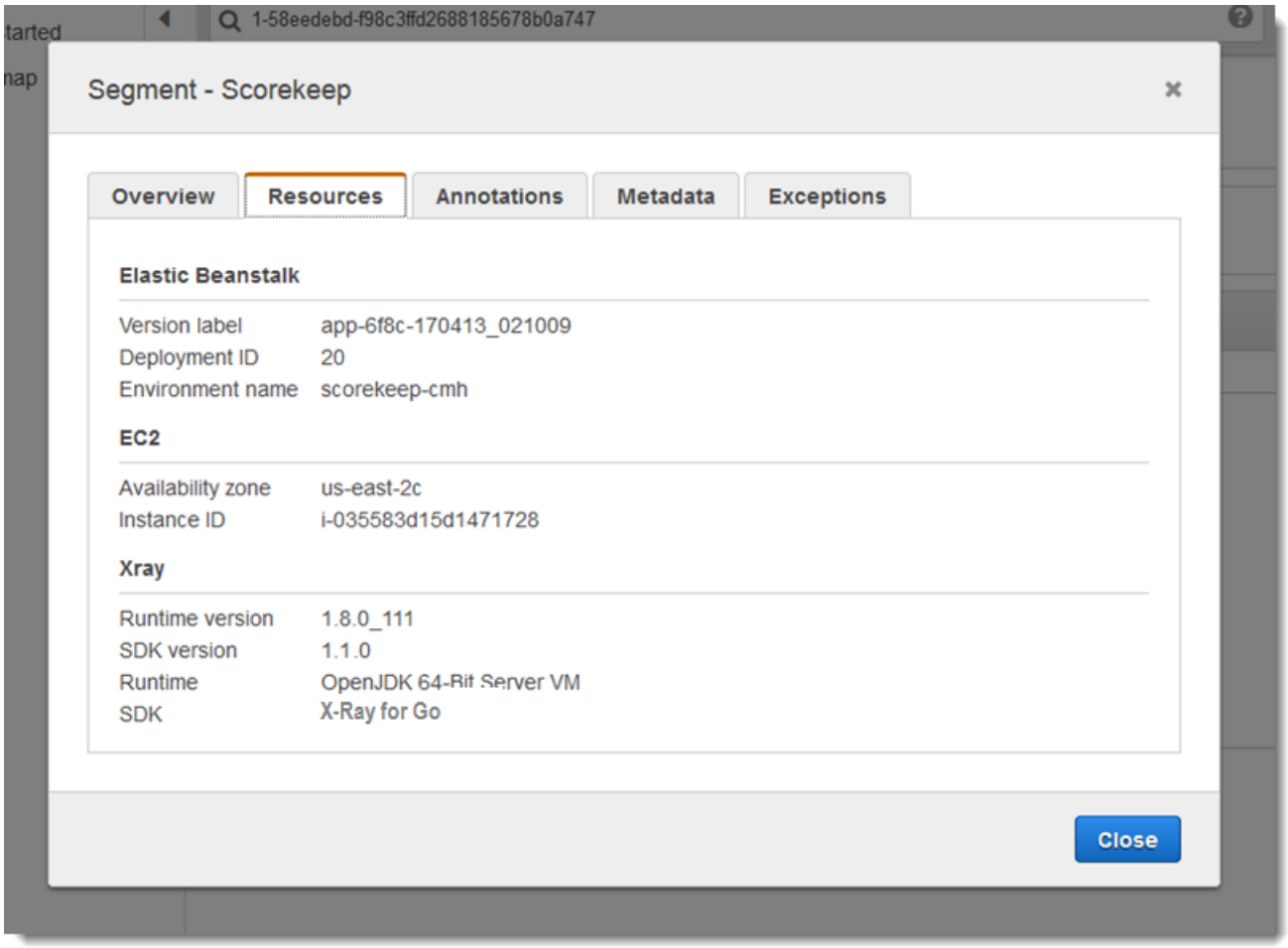
## Service plugins

Use `plugins` to record information about the service hosting your application.

**Plugins**

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.

- Amazon ECS – `ECSPlugin` adds the container ID.




To use a plugin, import one of the following packages.

```
"github.com/aws/aws-xray-sdk-go/awsplugins/ec2"
"github.com/aws/aws-xray-sdk-go/awsplugins/ecs"
"github.com/aws/aws-xray-sdk-go/awsplugins/beanstalk"
```

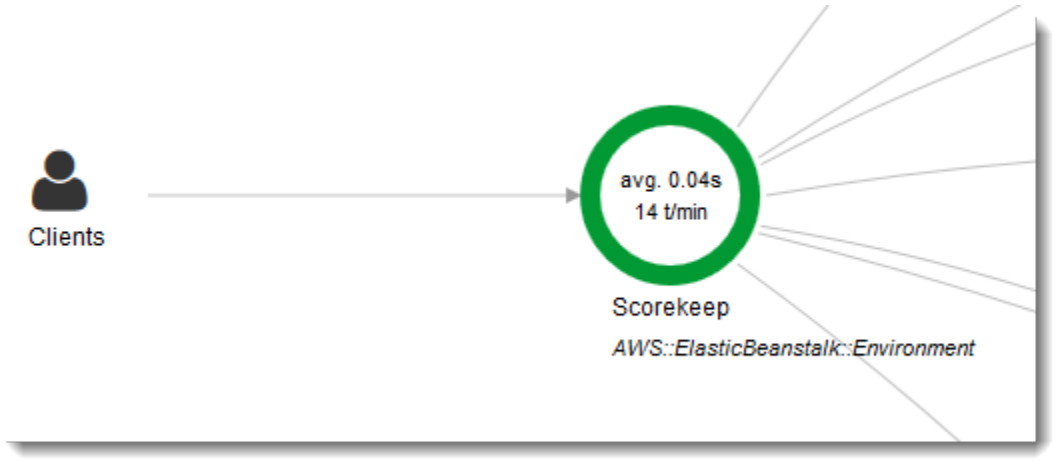
Each plugin has an explicit `Init()` function call that loads the plugin.

**Example ec2.Init()**

```
import (
 "os"

 "github.com/aws/aws-xray-sdk-go/awsplugins/ec2"
 "github.com/aws/aws-xray-sdk-go/xray"
 )

 func init() {
  // conditionally load plugin
  if os.Getenv("ENVIRONMENT") == "production" {
  ec2.Init()
 }
```

```
 xray.Configure(xray.Config{
  ServiceVersion: "1.2.3",

 })
}
```

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. The resource type appears under your application's name in the service map. For example, `AWS::ElasticBeanstalk::Environment`.




When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

# Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console (p. 70) to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

**Example sampling-rules.json**

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
```

```
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To provide backup rules, point to the local sampling JSON file by using `NewCentralizedStrategyWithFilePath`.

**Example main.go – Local sampling rule**

```
s, _ := sampling.NewCentralizedStrategyWithFilePath("sampling.json") // path to local
 sampling json
xray.Configure(xray.Config{SamplingStrategy: s})
```

To use only local rules, point to the local sampling JSON file by using `NewLocalizedStrategyFromFilePath`.

**Example main.go – Disable sampling**

```
s, _ := sampling.NewLocalizedStrategyFromFilePath("sampling.json") // path to local
 sampling json
xray.Configure(xray.Config{SamplingStrategy: s})
```

# Logging

> **Note**
> The `xray.Config{}` fields `LogLevel` and `LogFormat` are deprecated starting with version 1.0.0-rc.10.

X-Ray uses the following interface for logging. The default logger writes to `stdout` at `LogLevelInfo` and above.

```
type Logger interface {
 Log(level LogLevel, msg fmt.Stringer)
}

const (
 LogLevelDebug LogLevel = iota + 1
 LogLevelInfo
 LogLevelWarn
```

```
 LogLevelError
)
```

**Example write to `io.Writer`**

```
xray.SetLogger(xraylog.NewDefaultLogger(os.Stderr, xraylog.LogLevelError))
```

# Environment variables

You can use environment variables to configure the X-Ray SDK for Go. The SDK supports the following variables.

- `AWS_XRAY_TRACING_NAME` – Set the service name that the SDK uses for segments.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK sends trace data to `127.0.0.1:2000`. Use this variable if you have configured the daemon to listen on a different port (p. 140) or if it is running on a different host.
- `AWS_XRAY_CONTEXT_MISSING` – Set the value to determine how the SDK handles missing context errors. Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in the startup code when no request is open, or in code that spawns a new thread.
  - `RUNTIME_ERROR` – By default, the SDK is set to throw a runtime exception.
  - `LOG_ERROR` – Set to log an error and continue.

Environment variables override equivalent values set in code.

# Using configure

You can also configure the X-Ray SDK for Go using the `Configure` method. `Configure` takes one argument, a `Config` object, with the following, optional fields.

DaemonAddr

This string specifies the host and port of the X-Ray daemon listener. If not specified, X-Ray uses the value of the `AWS_XRAY_DAEMON_ADDRESS` environment variable. If that value is not set, it uses "127.0.0.1:2000".

ServiceVersion

This string specifies the version of the service. If not specified, X-Ray uses the empty string ("").

SamplingStrategy

This `SamplingStrategy` object specifies which of your application calls are traced. If not specified, X-Ray uses a `LocalizedSamplingStrategy`, which takes the strategy as defined in `xray/resources/DefaultSamplingRules.json`.

StreamingStrategy

This `StreamingStrategy` object specifies whether to stream a segment when **RequiresStreaming** returns **true**. If not specified, X-Ray uses a `DefaultStreamingStrategy` that streams a sampled segment if the number of subsegments is greater than 20.

ExceptionFormattingStrategy

This `ExceptionFormattingStrategy` object specifies how you want to handle various exceptions. If not specified, X-Ray uses a `DefaultExceptionFormattingStrategy` with an `XrayError` of type `error`, the error message, and stack trace.

# Instrumenting incoming HTTP requests with the X-Ray SDK for Go

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

Use `xray.Handler` to instrument incoming HTTP requests. The X-Ray SDK for Go implements the standard Go library `http.Handler` interface in the `xay.Handler` class to intercept web requests. The `xay.Handler` class wraps the provided `http.Handler` with `xray.Capture` using the request's context, parsing the incoming headers, adding response headers if needed, and sets HTTP-specific trace fields.

When you use this class to handle HTTP requests and responses, the X-Ray SDK for Go creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

> **Note**
> For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray (p. 167) for more information.

The following example intercepts requests on port 8000 and returns "Hello!" as a response. It creates the segment `myApp` and instruments calls through any application.

**Example main.go**

```
func main() {
  http.Handle("/", xray.Handler(xray.NewFixedSegmentNamer("MyApp"), http.HandlerFunc(func(w
 http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!"))
  })))

  http.ListenAndServe(":8000", nil)
}
```

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).

- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

## Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field (p. 103).

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you create the handler, as shown in the previous section.

> **Note**
> You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable (p. 179).

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request doesn't match the pattern. To name segments dynamically, use `NewDynamicSegmentNamer` to configure the default name and pattern to match.

### Example main.go

If the hostname in the request matches the pattern `*.example.com`, use the hostname. Otherwise, use `MyApp`.

```
func main() {
  http.Handle("/", xray.Handler(xray.NewDynamicSegmentNamer("MyApp", "*.example.com"),
 http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!"))
  })))

  http.ListenAndServe(":8000", nil)
}
```

# Tracing AWS SDK calls with the X-Ray SDK for Go

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Go tracks the calls downstream in subsegments (p. 183). Traced AWS services and

resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the service map in the X-Ray console.

To trace AWS SDK clients, wrap the client object with the `xray.AWS()` call as shown in the following example.

**Example main.go**

```
var dynamo *dynamodb.DynamoDB
func main() {
  dynamo = dynamodb.New(session.Must(session.NewSession()))
  xray.AWS(dynamo.Client)
}
```

Then, when you use the AWS SDK client, use the `withContext` version of the call method, and pass it the `context` from the `http.Request` object passed to the handler (p. 180).

**Example main.go – AWS SDK call**

```
func listTablesWithContext(ctx context.Context) {
  output := dynamo.ListTablesWithContext(ctx, &dynamodb.ListTablesInput{})
  doSomething(output)
}
```

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

**Example Subsegment for a call to DynamoDB to save an item**

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

# Tracing calls to downstream HTTP web services with the X-Ray SDK for Go

When your application makes calls to microservices or public HTTP APIs, you can use the `xray.Client` to instrument those calls as subsegments of your Go application, as shown in the following example, where *http-client* is an HTTP client.

Client creates a shallow copy of the provided http client, defaulting to `http.DefaultClient`, with roundtripper wrapped with `xray.RoundTripper`.

**Example main.go – HTTP client**

```
myClient := xray.Client(http-client)
```

# Tracing SQL queries with the X-Ray SDK for Go

To trace SQL calls to PostgreSQL or MySQL, replacing `sql.Open` calls to `xray.SQLContext`, as shown in the following example. Use URLs instead of configuration strings if possible.

**Example main.go**

```
func main() {
  db, err := xray.SQLContext("postgres", "postgres://user:password@host:port/db")
  row, err := db.QueryRowContext(ctx, "SELECT 1") // Use as normal
}
```

# Generating custom subsegments with the X-Ray SDK for Go

Subsegments extend a trace's segment (p. 21) with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

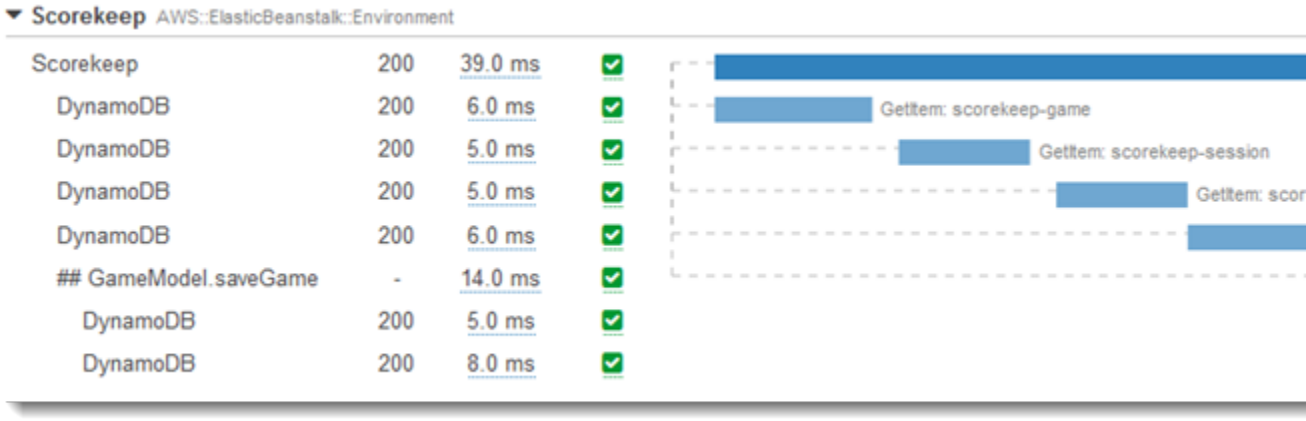
Use the `Capture` method to create a subsegment around a function.

**Example main.go – Custom subsegment**

```
func criticalSection(ctx context.Context) {
  //this is an example of a subsegment
  xray.Capture(ctx, "GameModel.saveGame", func(ctx1 context.Context) error {
    var err error

    section.Lock()
    result := someLockedResource.Go()
    section.Unlock()

    xray.AddMetadata(ctx1, "ResourceResult", result)
```

```
})
```

The following screenshot shows an example of how the `saveGame` subsegment might appear in traces for the application `Scorekeep`.




# Add annotations and metadata to segments with the X-Ray SDK for Go

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

**Annotations** are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions (p. 59). Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

**Metadata** are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings (p. 185) on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

**Sections**



## Recording annotations with the X-Ray SDK for Go

Use annotations to record information on segments that you want indexed for search.

**Annotation Requirements**

- **Keys** – Up to 500 alphanumeric characters. No spaces or symbols except underscores.
- **Values** – Up to 1,000 Unicode characters.
- **Entries** – Up to 50 annotations per trace.

To record annotations, call `AddAnnotation` with a string containing the metadata you want to associate with the segment.

```
xray.AddAnnotation(key string, value interface{})
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `AddAnnotation` twice with the same key overwrites previously recorded values on the same segment.

To find traces that have annotations with specific values, use the `annotations.key` keyword in a filter expression (p. 59).

# Recording metadata with the X-Ray SDK for Go

Use metadata to record information on segments that you don't need indexed for search.

To record metadata, call `AddMetadata` with a string containing the metadata you want to associate with the segment.

```
xray.AddMetadata(key string, value interface{})
```

# Recording user IDs with the X-Ray SDK for Go

Record user IDs on request segments to identify the user who sent the request.

**To record user IDs**

1. Get a reference to the current segment from `AWSXRay`.

   ```
   import (
     "context"
     "github.com/aws/aws-xray-sdk-go/xray"
   )

   mySegment := xray.GetSegment(context)
   ```

2. Call `setUser` with a String ID of the user who sent the request.

   ```
   mySegment.User = "U12345"
   ```

To find traces for a user ID, use the `user` keyword in a filter expression (p. 59).

# AWS X-Ray SDK for Java

The X-Ray SDK for Java is a set of libraries for Java web applications that provide classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an SQL database connector. You can also create segments manually and add debug information in annotations and metadata.

> **Note**
> The X-Ray SDK for Java is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-java

Start by adding `AWSXRayServletFilter` as a servlet filter (p. 196) to trace incoming requests. A servlet filter creates a segment (p. 21). While the segment is open you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

Starting in release 1.3, you can instrument your application using aspect-oriented programming (AOP) in Spring (p. 211). What this means is that you can instrument your application, while it is running on AWS, without adding any code to your application's runtime.

Next, use the X-Ray SDK for Java to instrument your AWS SDK for Java clients by including the SDK Instrumentor submodule (p. 188) in your build configuration. Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the service map to help you identify errors and throttling issues on individual connections.

If you don't want to instrument all downstream calls to AWS services, you can leave out the Instrumentor submodule and choose which clients to instrument. Instrument individual clients by adding a `TracingHandler` (p. 199) to an AWS SDK service client.

Other X-Ray SDK for Java submodules provide instrumentation for downstream calls to HTTP web APIs and SQL databases. You can use the X-Ray SDK for Java versions of `HTTPClient` and `HTTPClientBuilder` (p. 200) in the Apache HTTP submodule to instrument Apache HTTP clients. To instrument SQL queries, add the SDK's interceptor to your data source (p. 202).

Once you get going with the SDK, customize its behavior by configuring the recorder and servlet filter (p. 189). You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata (p. 205). Annotations are simple key-value pairs that are indexed for use with filter expressions (p. 59), so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

> **Annotations and Metadata**
> Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain many subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments (p. 203). You can create a custom

subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

# Submodules

You can download the X-Ray SDK for Java from Maven. The X-Ray SDK for Java is split into submodules by use case, with a bill of materials for version management:

- `aws-xray-recorder-sdk-core` (required) – Basic functionality for creating segments and transmitting segments. Includes `AWSXRayServletFilter` for instrumenting incoming requests.
- `aws-xray-recorder-sdk-aws-sdk` – Instruments calls to AWS services made with AWS SDK for Java clients by adding a tracing client as a request handler.
- `aws-xray-recorder-sdk-aws-sdk-v2` – Instruments calls to AWS services made with AWS SDK for Java 2.2 and later clients by adding a tracing client as a request intereceptor.
- `aws-xray-recorder-sdk-aws-sdk-instrumentor` – With `aws-xray-recorder-sdk-aws-sdk`, instruments all AWS SDK for Java clients automatically.
- `aws-xray-recorder-sdk-aws-sdk-v2-instrumentor` – With `aws-xray-recorder-sdk-aws-sdk`, instruments all AWS SDK for Java 2.2 and later clients automatically.
- `aws-xray-recorder-sdk-apache-http` – Instruments outbound HTTP calls made with Apache HTTP clients.
- `aws-xray-recorder-sdk-spring` – Provides interceptors for Spring AOP Framework applications.
- `aws-xray-recorder-sdk-sql-postgres` – Instruments outbound calls to a PostgreSQL database made with JDBC.
- `aws-xray-recorder-sdk-sql-mysql` – Instruments outbound calls to a MySQL database made with JDBC.
- `aws-xray-recorder-sdk-bom` – Provides a bill of materials that you can use to specify the version to use for all submodules.
- `aws-xray-recorder-sdk-metrics` – Publish unsampled Amazon CloudWatch metrics from your collected X-Ray segments.

If you use Maven or Gradle to build your application, add the X-Ray SDK for Java to your build configuration (p. 188).

For reference documentation of the SDK's classes and methods, see AWS X-Ray SDK for Java API Reference.

# Requirements

The X-Ray SDK for Java requires Java 8 or later, Servlet API 3, the AWS SDK, and Jackson.

The SDK depends on the following libraries at compile and runtime:

- AWS SDK for Java version 1.11.398 or later
- Servlet API 3.1.0

These dependencies are declared in the SDK's `pom.xml` file and are included automatically if you build using Maven or Gradle.

If you use a library that is included in the X-Ray SDK for Java, you must use the included version. For example, if you already depend on Jackson at runtime and include JAR files in your deployment for that

dependency, you must remove those JAR files because the SDK JAR includes its own versions of Jackson libraries.

# Dependency management

The X-Ray SDK for Java is available from Maven:

- **Group** – `com.amazonaws`
- **Artifact** – `aws-xray-recorder-sdk-bom`
- **Version** – `2.4.0`

If you use Maven to build your application, add the SDK as a dependency in your `pom.xml` file.

**Example pom.xml - dependencies**

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-xray-recorder-sdk-bom</artifactId>
      <version>2.4.0</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<dependencies>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-core</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-apache-http</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-aws-sdk</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-aws-sdk-instrumentor</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-sql-postgres</artifactId>
  </dependency>
  <dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-sql-mysql</artifactId>
  </dependency>
</dependencies>
```

For Gradle, add the SDK as a compile-time dependency in your `build.gradle` file.

**Example build.gradle - dependencies**

```
dependencies {
```

```
  compile("org.springframework.boot:spring-boot-starter-web")
  testCompile("org.springframework.boot:spring-boot-starter-test")
  compile("com.amazonaws:aws-java-sdk-dynamodb")
  compile("com.amazonaws:aws-xray-recorder-sdk-core")
  compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk")
  compile("com.amazonaws:aws-xray-recorder-sdk-aws-sdk-instrumentor")
  compile("com.amazonaws:aws-xray-recorder-sdk-apache-http")
  compile("com.amazonaws:aws-xray-recorder-sdk-sql-postgres")
  compile("com.amazonaws:aws-xray-recorder-sdk-sql-mysql")
  testCompile("junit:junit:4.11")
}
dependencyManagement {
    imports {
        mavenBom('com.amazonaws:aws-java-sdk-bom:1.11.39')
        mavenBom('com.amazonaws:aws-xray-recorder-sdk-bom:2.4.0')
    }
}
```

If you use Elastic Beanstalk to deploy your application, you can use Maven or Gradle to build on-instance each time you deploy, instead of building and uploading a large archive that includes all of your dependencies. See the sample application (p. 115) for an example that uses Gradle.

# Configuring the X-Ray SDK for Java

The X-Ray SDK for Java includes a class named `AWSXRay` that provides the global recorder. This is a `TracingHandler` that you can use to instrument your code. You can configure the global recorder to customize the `AWSXRayServletFilter` that creates segments for incoming HTTP calls.

**Sections**

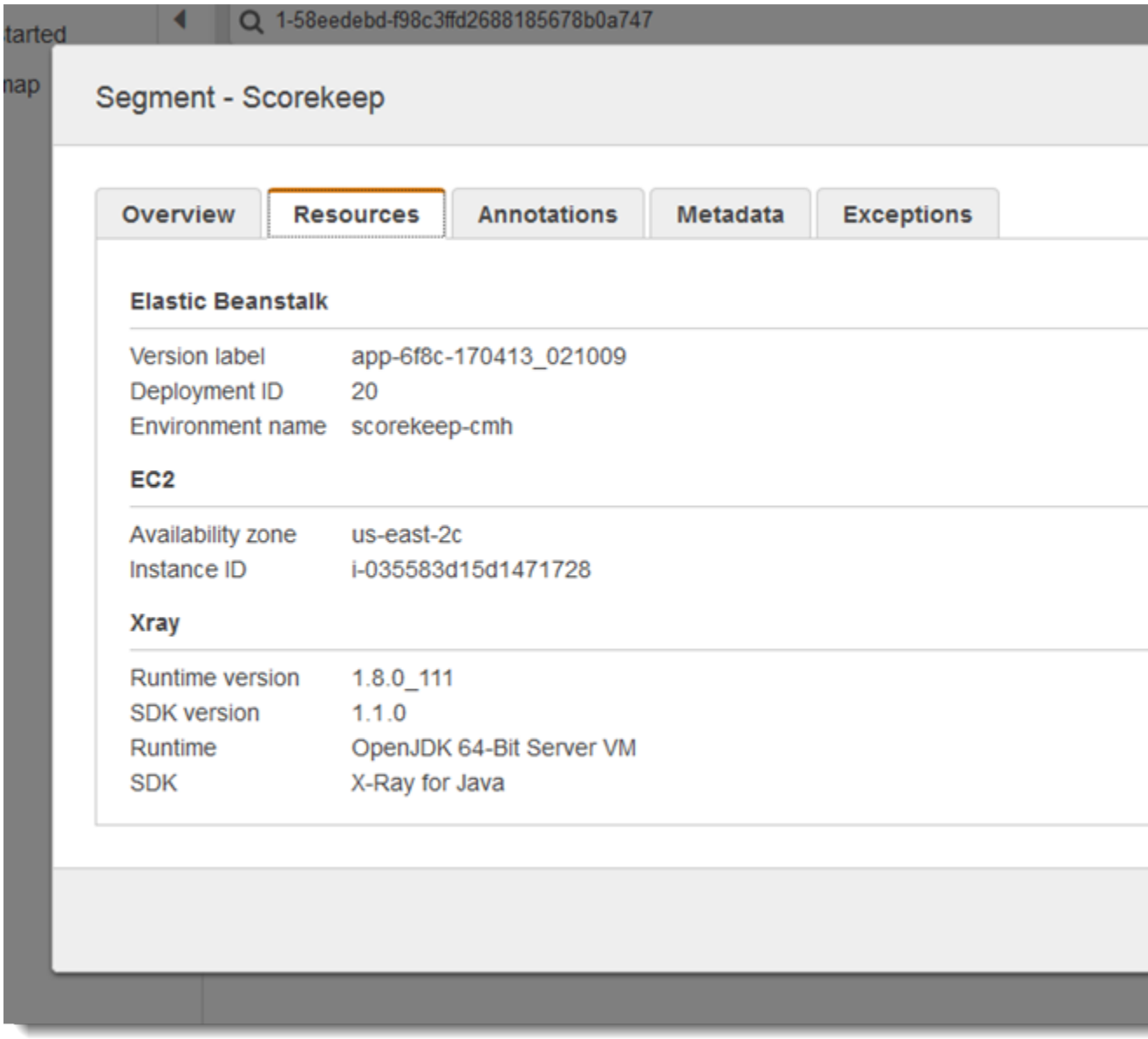


## Service plugins

Use `plugins` to record information about the service hosting your application.

**Plugins**

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.
- Amazon EKS – `EKSPlugin` adds the container ID, cluster name, pod ID, and the CloudWatch Logs Group.



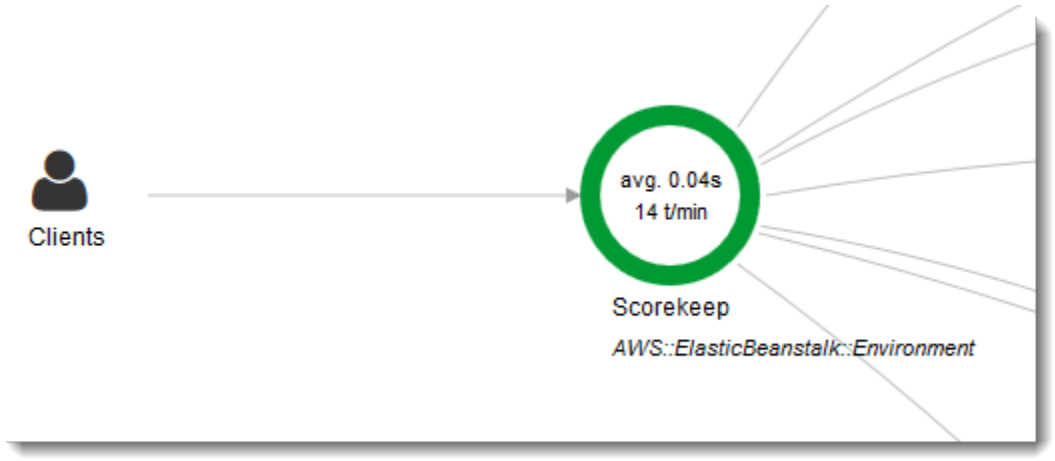

To use a plugin, call `withPlugin` on your `AWSXRayRecorderBuilder`.

**Example src/main/java/scorekeep/WebConfig.java - recorder**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {
...
  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin()).withPlugin(new ElasticBeanstalkPlugin());
```

```
    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
  }
}
```

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. The resource type appears under your application's name in the service map. For example, `AWS::ElasticBeanstalk::Environment`.




When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

# Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console (p. 70) to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

**Example sampling-rules.json**

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
```

```
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To provide backup rules in Spring, configure the global recorder with a `CentralizedSamplingStrategy` in a configuration class.

**Example src/main/java/myapp/WebConfig.java - recorder configuration**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {

  static {
  AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin());

  URL ruleFile = WebConfig.class.getResource("file://sampling-rules.json");
  builder.withSamplingStrategy(new CentralizedSamplingStrategy(ruleFile));

  AWSXRay.setGlobalRecorder(builder.build());
}
```

For Tomcat, add a listener that extends `ServletContextListener` and register the listener in the deployment descriptor.

**Example src/com/myapp/web/Startup.java**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

import java.net.URL;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
```

```
public class Startup implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent event) {
        AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder.standard().withPlugin(new
 EC2Plugin());

        URL ruleFile = Startup.class.getResource("/sampling-rules.json");
        builder.withSamplingStrategy(new CentralizedSamplingStrategy(ruleFile));

        AWSXRay.setGlobalRecorder(builder.build());
    }

    @Override
    public void contextDestroyed(ServletContextEvent event) { }
}
```

**Example WEB-INF/web.xml**

```
...
  <listener>
    <listener-class>com.myapp.web.Startup</listener-class>
  </listener>
```

To use local rules only, replace the `CentralizedSamplingStrategy` with a `LocalizedSamplingStrategy`.

```
builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));
```

# Logging

By default, the SDK outputs `SEVERE`-level and `ERROR`-level messages to your application logs. You can enable debug-level logging on the SDK to output more detailed logs to your application log file.

**Example application.properties**

Set the logging level with the `logging.level.com.amazonaws.xray` property.

```
logging.level.com.amazonaws.xray = DEBUG
```

Use debug logs to identify issues, such as unclosed subsegments, when you generate subsegments manually (p. 203).

## Trace ID injection into logs

To expose the current trace ID to your log statements, you can inject the ID into the mapped diagnostic context (MDC). Using the `SegmentListener` interface, methods are called from the X-Ray recorder during segment lifecycle events. When a segment begins, the trace ID is injected into the MDC with the key `AWS-XRAY-TRACE-ID`. When that segment ends, the key is removed from the MDC. This exposes the trace ID to the logging library in use.

This feature works with Java applications instrumented with the AWS X-Ray SDK for Java, and supports the following logging configurations:

- SLF4J front-end API with Logback backend
- SLF4J front-end API with Log4J2 backend

- Log4J2 front-end API with Log4J2 backend

See the following tabs for the needs of each front end and each backend.

SLF4J Frontend

1. Add the following Maven dependency to your project.

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-slf4j</artifactId>
    <version>2.4.0</version>
</dependency>
```

2. Include the `withSegmentListener` method when building the `AWSXRayRecorder`. This adds a `SegmentListener` class, which automatically injects new trace IDs into the SLF4J MDC.

   **Example `AWSXRayRecorderBuilder` Statement**

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new SLF4JSegmentListener());
```

Log4J2 front end

1. Add the following Maven dependency to your project.

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-xray-recorder-sdk-log4j</artifactId>
    <version>2.4.0</version>
</dependency>
```

2. Include the `withSegmentListener` method when building the `AWSXRayRecorder`. This will add a `SegmentListener` class, which automatically injects new trace IDs into the SLF4J MDC.

   **Example `AWSXRayRecorderBuilder` Statement**

```
AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
        .standard().withSegmentListener(new Log4JSegmentListener());
```

Logback backend

To insert the trace ID into your log events, you must modify the logger's `PatternLayout`, which formats each logging statement.

1. Find where the `patternLayout` is configured. You can do this programmatically, or through an XML configuration file. To learn more, see Logback configuration.
2. Insert `%X{AWS-XRAY-TRACE-ID}` anywhere in the `patternLayout` to insert the trace ID in future logging statements. `%X{}` indicates that you are retrieving a value with the provided key from the MDC. To learn more about PatternLayouts in Logback, see PatternLayout.

Log4J2 backend

1. Find where the `patternLayout` is configured. You can do this programmatically, or through a configuration file written in XML, JSON, YAML, or properties format.

To learn more about configuring Log4J2 through a configuration file, see Configuration.

To learn more about configuring Log4J2 programmatically, see Programmatic Configuration.

2. Insert `%X{AWS-XRAY-TRACE-ID}` anywhere in the `PatternLayout` to insert the trace ID in future logging statements. `%X{}` indicates that you are retrieving a value with the provided key from the MDC. To learn more about PatternLayouts in Log4J2, see Pattern Layout.

#### Trace ID Injection Example

The following shows a `PatternLayout` string modified to include the trace ID. The trace ID is printed after the thread name (`%t`) and before the log level (`%-5p`).

#### Example `PatternLayout` With ID injection

```
%d{HH:mm:ss.SSS} [%t] %X{AWS-XRAY-TRACE-ID} %-5p %m%n
```

AWS X-Ray automatically prints the key and the trace ID in the log statement for easy parsing. The following shows a log statement using the modified `PatternLayout`.

#### Example Log statement with ID injection

```
2019-09-10 18:58:30.844 [nio-5000-exec-4]  AWS-XRAY-TRACE-
ID: 1-5d77f256-19f12e4eaa02e3f76c78f46a WARN 1 - Your logging message here
```

The logging message itself is housed in the pattern `%m` and is set when calling the logger.

# Environment variables

You can use environment variables to configure the X-Ray SDK for Java. The SDK supports the following variables.

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the service name that you set on the servlet filter's segment naming strategy (p. 197).
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK uses `127.0.0.1:2000` for both trace data (UDP) and sampling (TCP). Use this variable if you have configured the daemon to listen on a different port (p. 140) or if it is running on a different host.

  **Format**

  - **Same port** – *`address`*`:`*`port`*
  - **Different ports** – `tcp:`*`address`*`:`*`port`* `udp:`*`address`*`:`*`port`*
- `AWS_XRAY_CONTEXT_MISSING` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.

  **Valid Values**

  - `RUNTIME_ERROR` – Throw a runtime exception (default).
  - `LOG_ERROR` – Log an error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

Environment variables override equivalent system properties (p. 196) and values set in code.

## System properties

You can use system properties as a JVM-specific alternative to environment variables (p. 195). The SDK supports the following properties:

- `com.amazonaws.xray.strategy.tracingName` – Equivalent to `AWS_XRAY_TRACING_NAME`.
- `com.amazonaws.xray.emitters.daemonAddress` – Equivalent to `AWS_XRAY_DAEMON_ADDRESS`.
- `com.amazonaws.xray.strategy.contextMissingStrategy` – Equivalent to `AWS_XRAY_CONTEXT_MISSING`.

If both a system property and the equivalent environment variable are set, the environment variable value is used. Either method overrides values set in code.

# Tracing incoming requests with the X-Ray SDK for Java

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

Use a `Filter` to instrument incoming HTTP requests. When you add the X-Ray servlet filter to your application, the X-Ray SDK for Java creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

> **Note**
> For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray (p. 167) for more information.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The message handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

**Sections**



# Adding a tracing filter to your application (Tomcat)

For Tomcat, add a `<filter>` to your project's `web.xml` file. Use the `fixedName` parameter to specify a service name (p. 197) to apply to segments created for incoming requests.

**Example WEB-INF/web.xml - Tomcat**

```
<filter>
  <filter-name>AWSXRayServletFilter</filter-name>
  <filter-class>com.amazonaws.xray.javax.servlet.AWSXRayServletFilter</filter-class>
  <init-param>
    <param-name>fixedName</param-name>
    <param-value>MyApp</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>AWSXRayServletFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

# Adding a tracing filter to your application (spring)

For Spring, add a `Filter` to your `WebConfig` class. Pass the segment name to the `AWSXRayServletFilter` constructor as a string.

**Example src/main/java/myapp/WebConfig.java - spring**

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;

@Configuration
public class WebConfig {

  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter("Scorekeep");
  }
}
```

# Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field (p. 103).

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from

naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you initialize the servlet filter, as shown in the previous section (p. 197). This has the same effect as creating a FixedSegmentNamingStrategy and passing it to `AWSXRayServletFilter` constructor.

> **Note**
> You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable (p. 195).

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request does not match the pattern. To name segments dynamically in Tomcat, use the `dynamicNamingRecognizedHosts` and `dynamicNamingFallbackName` to define the pattern and default name, respectively.

**Example WEB-INF/web.xml - servlet filter with dynamic naming**

```
<filter>
  <filter-name>AWSXRayServletFilter</filter-name>
  <filter-class>com.amazonaws.xray.javax.servlet.AWSXRayServletFilter</filter-class>
  <init-param>
    <param-name>dynamicNamingRecognizedHosts</param-name>
    <param-value>*.example.com</param-value>
  </init-param>
  <init-param>
    <param-name>dynamicNamingFallbackName</param-name>
    <param-value>MyApp</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>AWSXRayServletFilter</filter-name>
  <url-pattern>*</url-pattern>
</filter-mapping>
```

For Spring, create a DynamicSegmentNamingStrategy and pass it to the `AWSXRayServletFilter` constructor.

**Example src/main/java/myapp/WebConfig.java - servlet filter with dynamic naming**

```
package myapp;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Bean;
import javax.servlet.Filter;
import com.amazonaws.xray.javax.servlet.AWSXRayServletFilter;
import com.amazonaws.xray.strategy.DynamicSegmentNamingStrategy;

@Configuration
public class WebConfig {
```

```
  @Bean
  public Filter TracingFilter() {
    return new AWSXRayServletFilter(new DynamicSegmentNamingStrategy("MyApp",
 "*.example.com"));
  }
}
```

# Tracing AWS SDK calls with the X-Ray SDK for Java

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Java tracks the calls downstream in subsegments (p. 203). Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the service map in the X-Ray console.

The X-Ray SDK for Java automatically instruments all AWS SDK clients when you include the `aws-sdk` and an `aws-sdk-instrumentor` submodules (p. 187) in your build. If you don't include the Instrumentor submodule, you can choose to instrument some clients while excluding others.

To instrument individual clients, remove the `aws-sdk-instrumentor` submodule from your build and add an `XRayClient` as a `TracingHandler` on your AWS SDK client using the service's client builder.

For example, to instrument an `AmazonDynamoDB` client, pass a tracing handler to `AmazonDynamoDBClientBuilder`.

**Example MyModel.java - DynamoDB client**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.handlers.TracingHandler;

...
public class MyModel {
  private AmazonDynamoDB client = AmazonDynamoDBClientBuilder.standard()
        .withRegion(Regions.fromName(System.getenv("AWS_REGION")))
        .withRequestHandlers(new TracingHandler(AWSXRay.getGlobalRecorder()))
        .build();
...
```

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

**Example Subsegment for a call to DynamoDB to save an item**

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
```

```
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

To instrument downstream calls to AWS services with AWS SDK for Java 2.2 and later, you can omit the `aws-xray-recorder-sdk-aws-sdk-v2-instrumentor` module from your build configuration. Include the `aws-xray-recorder-sdk-aws-sdk-v2 module` instead, then instrument individual clients by configuring them with a `TracingInterceptor`.

**Example AWS SDK for Java 2.2 and later - tracing interceptor**

```
import com.amazonaws.xray.interceptors.TracingInterceptor;
import software.amazon.awssdk.core.client.config.ClientOverrideConfiguration
import software.amazon.awssdk.services.dynamodb.DynamoDbClient;
//...
public class MyModel {
private DynamoDbClient client = DynamoDbClient.builder()
.region(Region.US_WEST_2)
.overrideConfiguration(ClientOverrideConfiguration.builder()
.addExecutionInterceptor(new TracingInterceptor())
.build()
)
.build();
//...
```

# Tracing calls to downstream HTTP web services with the X-Ray SDK for Java

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for Java's version of `HttpClient` to instrument those calls and add the API to the service graph as a downstream service.

The X-Ray SDK for Java includes `DefaultHttpClient` and `HttpClientBuilder` classes that can be used in place of the Apache HttpComponents equivalents to instrument outgoing HTTP calls.

- `com.amazonaws.xray.proxies.apache.http.DefaultHttpClient` - `org.apache.http.impl.client.DefaultHttpClient`
- `com.amazonaws.xray.proxies.apache.http.HttpClientBuilder` - `org.apache.http.impl.client.HttpClientBuilder`

These libraries are in the `aws-xray-recorder-sdk-apache-http` (p. 186) submodule.

You can replace your existing import statements with the X-Ray equivalent to instrument all clients, or use the fully qualified name when you initialize a client to instrument specific clients.

### Example HttpClientBuilder

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.http.HttpEntity;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.util.EntityUtils;
import com.amazonaws.xray.proxies.apache.http.HttpClientBuilder;
...
  public String randomName() throws IOException {
    CloseableHttpClient httpclient = HttpClientBuilder.create().build();
    HttpGet httpGet = new HttpGet("http://names.example.com/api/");
    CloseableHttpResponse response = httpclient.execute(httpGet);
    try {
      HttpEntity entity = response.getEntity();
      InputStream inputStream = entity.getContent();
      ObjectMapper mapper = new ObjectMapper();
      Map<String, String> jsonMap = mapper.readValue(inputStream, Map.class);
      String name = jsonMap.get("name");
      EntityUtils.consume(entity);
      return name;
    } finally {
      response.close();
    }
  }
```

When you instrument a call to a downstream web api, the X-Ray SDK for Java records a subsegment with information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

### Example Subsegment for a downstream HTTP call

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

### Example Inferred segment for a downstream HTTP call

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-5880168b-fd5153bb58284b67678aa78c",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
```

```
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

# Tracing SQL queries with the X-Ray SDK for Java

Instrument SQL database queries by adding the X-Ray SDK for Java JDBC interceptor to your data source configuration.

- **PostgreSQL** – `com.amazonaws.xray.sql.postgres.TracingInterceptor`
- **MySQL** – `com.amazonaws.xray.sql.mysql.TracingInterceptor`

These interceptors are in the `aws-xray-recorder-sql-postgres` and `aws-xray-recorder-sql-mysql` submodules (p. 186), respectively. They implement `org.apache.tomcat.jdbc.pool.JdbcInterceptor` and are compatible with Tomcat connection pools.

For Spring, add the interceptor in a properties file and build the data source with Spring Boot's `DataSourceBuilder`.

**Example `src/main/java/resources/application.properties` - PostgreSQL JDBC interceptor**

```
spring.datasource.continue-on-error=true
spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=create-drop
spring.datasource.jdbc-interceptors=com.amazonaws.xray.sql.postgres.TracingInterceptor
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQL94Dialect
```

**Example `src/main/java/myapp/WebConfig.java` - Data source**

```
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.jdbc.DataSourceBuilder;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

import javax.servlet.Filter;
import javax.sql.DataSource;
import java.net.URL;

@Configuration
@EnableAutoConfiguration
@EnableJpaRepositories("myapp")
public class RdsWebConfig {

  @Bean
  @ConfigurationProperties(prefix = "spring.datasource")
  public DataSource dataSource() {
      logger.info("Initializing PostgreSQL datasource");
```

```
      return DataSourceBuilder.create()
              .driverClassName("org.postgresql.Driver")
              .url("jdbc:postgresql://" + System.getenv("RDS_HOSTNAME") + ":" +
 System.getenv("RDS_PORT") + "/ebdb")
              .username(System.getenv("RDS_USERNAME"))
              .password(System.getenv("RDS_PASSWORD"))
              .build();
  }
...
}
```

For Tomcat, call `setJdbcInterceptors` on the JDBC data source with a reference to the X-Ray SDK for Java class.

**Example `src/main/myapp/model.java` - Data source**

```
import org.apache.tomcat.jdbc.pool.DataSource;
...
DataSource source = new DataSource();
source.setUrl(url);
source.setUsername(user);
source.setPassword(password);
source.setDriverClassName("com.mysql.jdbc.Driver");
source.setJdbcInterceptors("com.amazonaws.xray.sql.mysql.TracingInterceptor;");
```

The Tomcat JDBC Data Source library is included in the X-Ray SDK for Java, but you can declare it as a provided dependency to document that you use it.

**Example `pom.xml` - JDBC data source**

```
<dependency>
  <groupId>org.apache.tomcat</groupId>
  <artifactId>tomcat-jdbc</artifactId>
  <version>8.0.36</version>
  <scope>provided</scope>
</dependency>
```

# Generating custom subsegments with the X-Ray SDK for Java

Subsegments extend a trace's segment (p. 21) with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

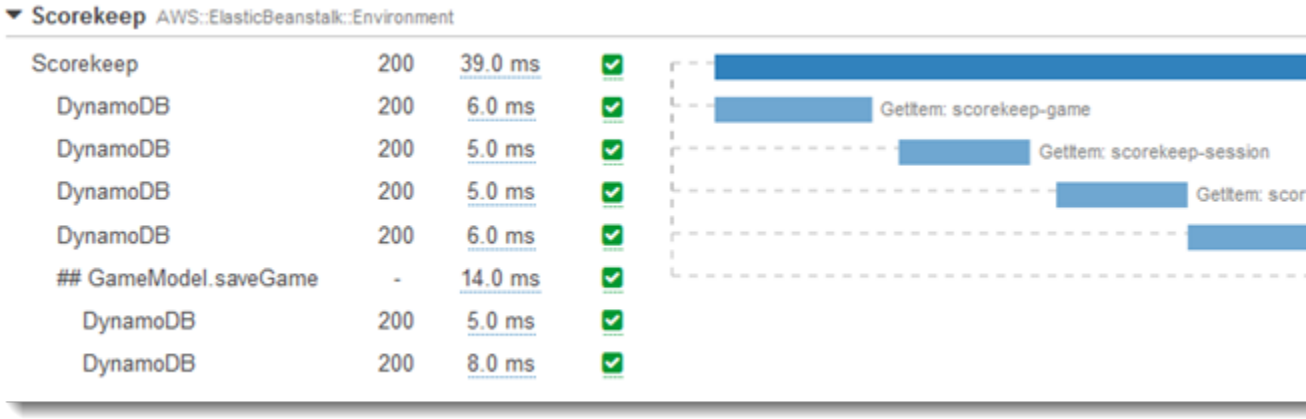
To manage subsegments, use the `beginSubsegment` and `endSubsegment` methods.

**Example GameModel.java - custom subsegment**

```
import com.amazonaws.xray.AWSXRay;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("Save Game");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
```

```
        throw new SessionNotFoundException(sessionId);
      }
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

In this example, the code within the subsegment loads the game's session from DynamoDB with a method on the session model, and uses the AWS SDK for Java's DynamoDB mapper to save the game. Wrapping this code in a subsegment makes the calls DynamoDB children of the `Save Game` subsegment in the trace view in the console.




If the code in your subsegment throws checked exceptions, wrap it in a `try` block and call `AWSXRay.endSubsegment()` in a `finally` block to ensure that the subsegment is always closed. If a subsegment is not closed, the parent segment cannot be completed and won't be sent to X-Ray.

For code that doesn't throw checked exceptions, you can pass the code to `AWSXRay.CreateSubsegment` as a Lambda function.

### Example Subsegment Lambda function

```
import com.amazonaws.xray.AWSXRay;

AWSXRay.createSubsegment("getMovies", (subsegment) -> {
    // function code
});
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for Java generates an ID for it and records the start time and end time.

### Example Subsegment with metadata

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
```

```
},
```

# Add annotations and metadata to segments with the X-Ray SDK for Java

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

**Annotations** are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions (p. 59). Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

**Metadata** are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings (p. 207) on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

**Sections**



## Recording annotations with the X-Ray SDK for Java

Use annotations to record information on segments or subsegments that you want indexed for search.

### Annotation Requirements

- **Keys** – Up to 500 alphanumeric characters. No spaces or symbols except underscores.
- **Values** – Up to 1,000 Unicode characters.
- **Entries** – Up to 50 annotations per trace.

### To record annotations

1. Get a reference to the current segment or subsegment from `AWSXRay`.

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Segment;
   ...
   Segment document = AWSXRay.getCurrentSegment();
   ```

   or

   ```
   import com.amazonaws.xray.AWSXRay;
   import com.amazonaws.xray.entities.Subsegment;
   ...
   Subsegment document = AWSXRay.getCurrentSubsegment();
   ```

2. Call `putAnnotation` with a String key, and a Boolean, Number, or String value.

```
document.putAnnotation("mykey", "my value");
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `putAnnotation` twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotations.`*`key`* keyword in a filter expression (p. 59).

**Example `src/main/java/scorekeep/GameModel.java` – Annotations and metadata**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

# Recording metadata with the X-Ray SDK for Java

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any object that can be serialized into a JSON object or array.

**To record metadata**

1. Get a reference to the current segment or subsegment from `AWSXRay`.

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
...
Segment document = AWSXRay.getCurrentSegment();
```

or

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Subsegment;
...
Subsegment document = AWSXRay.getCurrentSubsegment();
```

2. Call `putMetadata` with a String namespace, String key, and a Boolean, Number, String, or Object value.

```
document.putMetadata("my namespace", "my key", "my value");
```

   or

   Call `putMetadata` with just a key and value.

```
document.putMetadata("my key", "my value");
```

If you don't specify a namespace, the SDK uses `default`. Calling `putMetadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

**Example `src/main/java/scorekeep/GameModel.java` – Annotations and metadata**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
import com.amazonaws.xray.entities.Subsegment;
...
  public void saveGame(Game game) throws SessionNotFoundException {
    // wrap in subsegment
    Subsegment subsegment = AWSXRay.beginSubsegment("## GameModel.saveGame");
    try {
      // check session
      String sessionId = game.getSession();
      if (sessionModel.loadSession(sessionId) == null ) {
        throw new SessionNotFoundException(sessionId);
      }
      Segment segment = AWSXRay.getCurrentSegment();
      subsegment.putMetadata("resources", "game", game);
      segment.putAnnotation("gameid", game.getId());
      mapper.save(game);
    } catch (Exception e) {
      subsegment.addException(e);
      throw e;
    } finally {
      AWSXRay.endSubsegment();
    }
  }
```

# Recording user IDs with the X-Ray SDK for Java

Record user IDs on request segments to identify the user who sent the request.

**To record user IDs**

1. Get a reference to the current segment from `AWSXRay`.

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.entities.Segment;
...
Segment document = AWSXRay.getCurrentSegment();
```

2. Call `setUser` with a string ID of the user who sent the request.

```
document.setUser("U12345");
```

You can call `setUser` in your controllers to record the user ID as soon as your application starts processing a request. If you will only use the segment to set the user ID, you can chain the calls in a single line.

**Example src/main/java/scorekeep/MoveController.java – User ID**

```
import com.amazonaws.xray.AWSXRay;
...
  @RequestMapping(value="/{userId}", method=RequestMethod.POST)
  public Move newMove(@PathVariable String sessionId, @PathVariable String gameId,
 @PathVariable String userId, @RequestBody String move) throws SessionNotFoundException,
 GameNotFoundException, StateNotFoundException, RulesException {
    AWSXRay.getCurrentSegment().setUser(userId);
    return moveFactory.newMove(sessionId, gameId, userId, move);
  }
```

To find traces for a user ID, use the `user` keyword in a filter expression (p. 59).

# AWS X-Ray metrics for the X-Ray SDK for Java

This topic describes the AWS X-Ray namespace, metrics, and dimensions. You can use the X-Ray SDK for Java to publish unsampled Amazon CloudWatch metrics from your collected X-Ray segments. These metrics are derived from the segment's start and end time, and the error, fault, and throttled status flags. Use these trace metrics to expose retries and dependency issues within subsegments.

CloudWatch is essentially a metrics repository. A metric is the fundamental concept in CloudWatch and represents a time-ordered set of data points. You (or AWS services) publish metrics data points into CloudWatch and you retrieve statistics about those data points as an ordered set of time-series data.

Metrics are uniquely defined by a name, a namespace, and one or more dimensions. Each data point has a timestamp and, optionally, a unit of measure. When you request statistics, the returned data stream is identified by namespace, metric name, and dimension.

For more information about CloudWatch, see the *Amazon CloudWatch User Guide*.

## X-Ray CloudWatch metrics

The `ServiceMetrics/SDK` namespace includes the following metrics.

| Metric | Statistics available | Description | Units |
|---|---|---|---|
| `Latency` | Average, Minimum, Maximum, Count | The difference between the start and end time. Average, minimum, and maximum all describe operational latency. Count describes call count. | Milliseconds |
| `ErrorRate` | Average, Sum | The rate of requests that failed with a `4xx Client Error` status code, resulting in an error. | Percent |

| Metric | Statistics available | Description | Units |
|---|---|---|---|
| `FaultRate` | Average, Sum | The rate of traces that failed with a `5xx Server Error` status code, resulting in a fault. | Percent |
| `ThrottleRate` | Average, Sum | The rate of throttled traces that return a `419` status code. This is a subset of the `ErrorRate` metric. | Percent |
| `OkRate` | Average, Sum | The rate of traced requests resulting in an `OK` status code. | Percent |

# X-Ray CloudWatch dimensions

Use the dimensions in the following table to refine the metrics returned for your X-Ray instrumented Java applications.

| Dimension | Description |
|---|---|
| `ServiceType` | The type of the service, for example, `AWS::EC2::Instance` or `NONE`, if not known. |
| `ServiceName` | The canonical name for the service. |

# Enable X-Ray CloudWatch metrics

Use the following procedure to enable trace metrics in your instrumented Java application.

**To configure trace metrics**

1. Add the `aws-xray-recorder-sdk-metrics` package as a Maven dependency. For more information, see X-Ray SDK for Java Submodules (p. 187).
2. Enable a new `MetricsSegmentListener()` as part of the global recorder build.

   **Example src/com/myapp/web/Startup.java**

```
import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.plugins.ElasticBeanstalkPlugin;
import com.amazonaws.xray.strategy.sampling.LocalizedSamplingStrategy;

@Configuration
public class WebConfig {
...
  static {
    AWSXRayRecorderBuilder builder = AWSXRayRecorderBuilder
                                       .standard()
```

```
                                        .withPlugin(new EC2Plugin())
                                        .withPlugin(new ElasticBeanstalkPlugin())
                                        .withSegmentListener(new
 MetricsSegmentListener());

    URL ruleFile = WebConfig.class.getResource("/sampling-rules.json");
    builder.withSamplingStrategy(new LocalizedSamplingStrategy(ruleFile));

    AWSXRay.setGlobalRecorder(builder.build());
  }
}
```

3. Deploy the CloudWatch agent to collect metrics using Amazon Elastic Compute Cloud (Amazon EC2), Amazon Elastic Container Service (Amazon ECS), or Amazon Elastic Kubernetes Service (Amazon EKS):

   - To configure Amazon EC2, see Deploying the CloudWatch Agent and the X-Ray Daemon on Amazon EC2.
   - To configure Amazon ECS, see Deploying the CloudWatch Agent and the X-Ray Daemon on Amazon ECS.
   - To configure Amazon EKS, see Deploying the CloudWatch Agent and the X-Ray Daemon on Amazon EKS.

4. Configure the SDK to communicate with the CloudWatch agent. By default, the SDK communicates with the CloudWatch agent on the address `127.0.0.1`. You can configure alternate addresses by setting the environment variable or Java property to `address:port`.

   **Example Environment variable**

   ```
   AWS_XRAY_METRICS_DAEMON_ADDRESS=address:port
   ```

   **Example Java property**

   ```
   com.amazonaws.xray.metrics.daemonAddress=address:port
   ```

**To validate configuration**

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Open the **Metrics** tab to observe the influx of your metrics.
3. (Optional) In the CloudWatch console, on the **Logs** tab, open the `ServiceMetricsSDK` log group. Look for a log stream that matches the host metrics, and confirm the log messages.

# Passing segment context between threads in a multithreaded application

When you create a new thread in your application, the `AWSXRayRecorder` doesn't maintain a reference to the current segment or subsegment Entity. If you use an instrumented client in the new thread, the SDK tries to write to a segment that doesn't exist, causing a SegmentNotFoundException.

To avoid throwing exceptions during development, you can configure the recorder with a ContextMissingStrategy that tells it to log an error instead. You can configure the strategy in code with

SetContextMissingStrategy, or configure equivalent options with an environment variable (p. 195) or system property (p. 196).

One way to address the error is to use a new segment by calling beginSegment when you start the thread and endSegment when you close it. This works if you are instrumenting code that doesn't run in response to an HTTP request, like code that runs when your application starts.

If you use multiple threads to handle incoming requests, you can pass the current segment or subsegment to the new thread and provide it to the global recorder. This ensures that the information recorded within the new thread is associated with the same segment as the rest of the information recorded about that request.

To pass trace context between threads, call GetTraceEntity on the global recorder to get a reference to the current entity (segment or subsegment). Pass the entity to the new thread, and then call SetTraceEntity to configure the global recorder to use it to record trace data within the thread.

See Using instrumented clients in worker threads (p. 134) for an example.

# AOP with spring and the X-Ray SDK for Java

This topic describes how to use the X-Ray SDK and the Spring Framework to instrument your application without changing its core logic. This means that there is now a non-invasive way to instrument your applications running remotely in AWS.

You must perform three tasks to enable this feature.

**To enable AOP in spring**

1. Configure Spring (p. 211)
2. Annotate your code or implement an interface (p. 211)
3. Activate X-Ray in your application (p. 212)

## Configuring spring

You can use Maven or Gradle to configure Spring to use AOP to instrument your application.

If you use Maven to build your application, add the following dependency in your `pom.xml` file.

```
<dependency>
     <groupId>com.amazonaws</groupId>
     <artifactId>aws-xray-recorder-sdk-spring</artifactId>
     <version>2.4.0</version>
</dependency>
```

For Gradle, add the following dependency in your `build.gradle` file.

```
compile 'com.amazonaws:aws-xray-recorder-sdk-spring:2.4.0'
```

## Annotating your code or implementing an interface

Your classes must either be annotated with the `@XRayEnabled` annotation, or implement the `XRayTraced` interface. This tells the AOP system to wrap the functions of the affected class for X-Ray instrumentation.

# Activating x-ray in your application

To activate X-Ray tracing in your application, your code must extend the abstract class `AbstractXRayInterceptor` by overriding the following methods.

- `generateMetadata`—This function allows customization of the metadata attached to the current function's trace. By default, the class name of the executing function is recorded in the metadata. You can add more data if you need additional insights.
- `xrayEnabledClasses`—This function is empty, and should remain so. It serves as the host for a pointcut instructing the interceptor about which methods to wrap. Define the pointcut by specifying which of the classes that are annotated with `@XRayEnabled` to trace. The following pointcut statement tells the interceptor to wrap all controller beans annotated with the `@XRayEnabled` annotation.

```
@Pointcut("@within(com.amazonaws.xray.spring.aop.XRayEnabled) && bean(*Controller)")
```

# Example

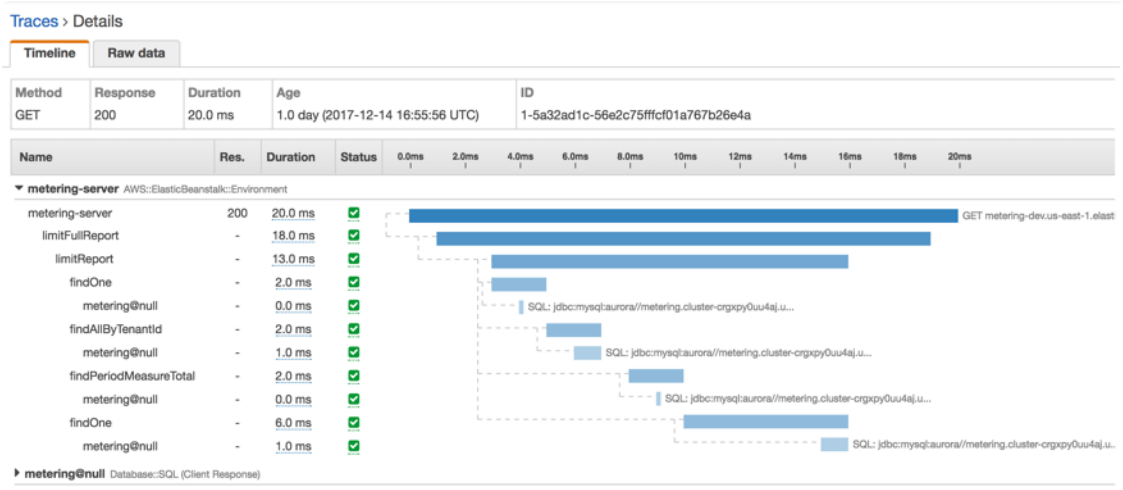
The following code extends the abstract class `AbstractXRayInterceptor`.

```
@Aspect
@Component
public class XRayInspector extends AbstractXRayInterceptor {
    @Override
    protected Map<String, Map<String, Object>> generateMetadata(ProceedingJoinPoint
 proceedingJoinPoint, Subsegment subsegment) throws Exception {
        return super.generateMetadata(proceedingJoinPoint, subsegment);
    }

  @Override
  @Pointcut("@within(com.amazonaws.xray.spring.aop.XRayEnabled) && bean(*Controller)")
  public void xrayEnabledClasses() {}

}
```

The following code is a class that will be instrumented by X-Ray.

```
@Service
@XRayEnabled
public class MyServiceImpl implements MyService {
    private final MyEntityRepository myEntityRepository;

    @Autowired
    public MyServiceImpl(MyEntityRepository myEntityRepository) {
        this.myEntityRepository = myEntityRepository;
    }

    @Transactional(readOnly = true)
    public List<MyEntity> getMyEntities(){
        try(Stream<MyEntity> entityStream = this.myEntityRepository.streamAll()){

            return entityStream.sorted().collect(Collectors.toList());
        }
    }
}
```

If you've configured your application correctly, you should see the complete call stack of the application, from the controller down through the service calls, as shown in the following screen shot of the console.



# The X-Ray SDK for Node.js

The X-Ray SDK for Node.js is a library for Express web applications and Node.js Lambda functions that provides classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK or HTTP clients.

> **Note**
> The X-Ray SDK for Node.js is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-node

If you use Express, start by adding the SDK as middleware (p. 219) on your application server to trace incoming requests. The middleware creates a segment (p. 21) for each traced request, and completes the segment when the response is sent. While the segment is open you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header (p. 26) and traces sampled requests automatically. For other functions, you can configure Lambda (p. 167) to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

> **Note**
> On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray (p. 167) for more information.

Next, use the X-Ray SDK for Node.js to instrument your AWS SDK for JavaScript in Node.js clients (p. 222). Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the service map to help you identify errors and throttling issues on individual connections.

The X-Ray SDK for Node.js also provides instrumentation for downstream calls to HTTP web APIs and SQL queries. Wrap your HTTP client in the SDK's capture method (p. 223) to record information about outgoing HTTP calls. For SQL clients, use the capture method for your database type (p. 224).

The middleware applies sampling rules to incoming requests to determine which requests to trace. You can configure the X-Ray SDK for Node.js (p. 216) to adjust the sampling behavior or to record information about the AWS compute resources on which your application runs.

Record additional information about requests and the work that your application does in annotations and metadata (p. 227). Annotations are simple key-value pairs that are indexed for use with filter expressions (p. 59), so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

> **Annotations and Metadata**
> Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments (p. 225). You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation about the SDK's classes and methods, see the AWS X-Ray SDK for Node.js API Reference.

# Requirements

The X-Ray SDK for Node.js requires Node.js and the following libraries:

- `atomic-batcher` – 1.0.2
- `cls-hooked` – 4.2.2
- `pkginfo` – 0.4.0
- `semver` – 5.3.0

The SDK pulls these libraries in when you install it with NPM.

To trace AWS SDK clients, the X-Ray SDK for Node.js requires a minimum version of the AWS SDK for JavaScript in Node.js.

- `aws-sdk` – 2.7.15

# Dependency management

The X-Ray SDK for Node.js is available from NPM.

- **Package** – `aws-xray-sdk`

For local development, install the SDK in your project directory with npm.

```
~/nodejs-xray$ npm install aws-xray-sdk
aws-xray-sdk@3.0.0
### aws-xray-sdk-core@3.0.0
# ### atomic-batcher@1.0.2
# ### cls-hooked@4.2.2
# # ### async-hook-jl@1.7.6
# # # ### stack-chain@1.3.7
# # ### emitter-listener@1.1.2
# # ### shimmer@1.2.1
# ### pkginfo@0.4.1
# ### semver@5.7.1
### aws-xray-sdk-express@3.0.0
### aws-xray-sdk-mysql@3.0.0
### aws-xray-sdk-postgres@3.0.0
```

Use the `--save` option to save the SDK as a dependency in your application's `package.json`.

```
~/nodejs-xray$ npm install aws-xray-sdk --save
aws-xray-sdk@3.0.0
```

# Node.js samples

Work with the AWS X-Ray SDK for Node.js to get an end-to-end view of requests as they travel through your Node.js applications.

- Node.js sample application on GitHub.

# Configuring the X-Ray SDK for Node.js

You can configure the X-Ray SDK for Node.js with plugins to include information about the service that your application runs on, modify the default sampling behavior, or add sampling rules that apply to requests to specific paths.

**Sections**



## Service plugins

Use `plugins` to record information about the service hosting your application.

**Plugins**

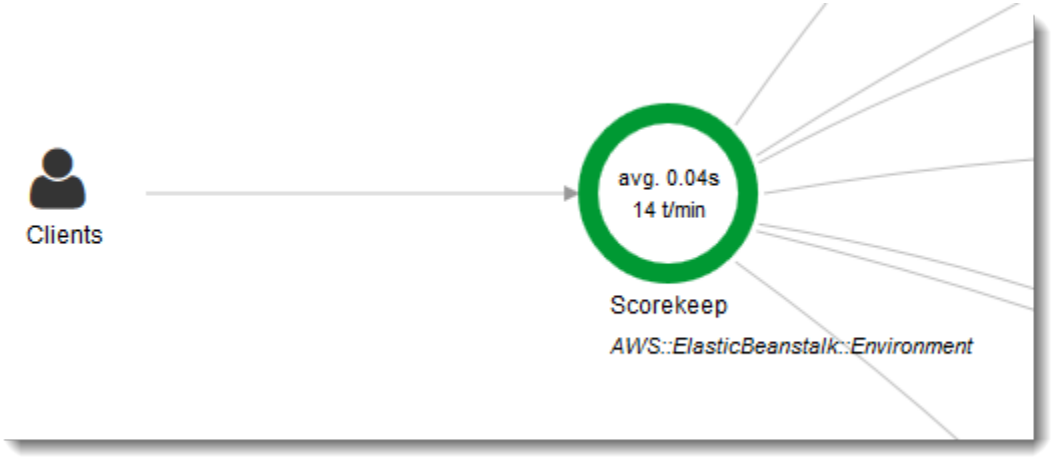
- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.

To use a plugin, configure the X-Ray SDK for Node.js client by using the `config` method.

**Example app.js - plugins**

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.config([AWSXRay.plugins.EC2Plugin,AWSXRay.plugins.ElasticBeanstalkPlugin]);
```

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. The resource type appears under your application's name in the service map. For example, `AWS::ElasticBeanstalk::Environment`.



When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

# Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console (p. 70) to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

**Example sampling-rules.json**

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To configure backup rules, tell the X-Ray SDK for Node.js to load sampling rules from a file with `setSamplingRules`.

**Example app.js - sampling rules from a file**

```
var AWSXRay = require('aws-xray-sdk');
```

```
AWSXRay.middleware.setSamplingRules('sampling-rules.json');
```

You can also define your rules in code and pass them to `setSamplingRules` as an object.

**Example app.js - sampling rules from an object**

```
var AWSXRay = require('aws-xray-sdk');
var rules = {
  "rules": [ { "description": "Player moves.", "service_name": "*", "http_method": "*",
 "url_path": "/api/move/*", "fixed_target": 0, "rate": 0.05 } ],
  "default": { "fixed_target": 1, "rate": 0.1 },
  "version": 1
  }

AWSXRay.middleware.setSamplingRules(rules);
```

To use only local rules, call `disableCentralizedSampling`.

```
AWSXRay.middleware.disableCentralizedSampling()
```

# Logging

To log output from the SDK, call `AWSXRay.setLogger(logger)`, where `logger` is an object that provides standard logging methods (`warn`, `info`, etc.).

**Example app.js - logging**

```
var AWSXRay = require('aws-xray-sdk');

// Create your own logger, or instantiate one using a library.
var logger = {
  error: (message, meta) => { /* logging code */ },
  warn: (message, meta) => { /* logging code */ },
  info: (message, meta) => { /* logging code */ },
  debug: (message, meta) => { /* logging code */ }
}

AWSXRay.setLogger(logger);
AWSXRay.config([AWSXRay.plugins.EC2Plugin]);
```

Call `setLogger` before you run other configuration methods to ensure that you capture output from those operations.

For a list of valid log level values, view the details in the Environment variables (p. 219) section.

To configure the SDK to output logs to the console without using a logging library, use the `AWS_XRAY_DEBUG_MODE` environment variable.

# X-Ray daemon address

If the X-Ray daemon listens on a port or host other than `127.0.0.1:2000`, you can configure the X-Ray SDK for Node.js to send trace data to a different address.

```
AWSXRay.setDaemonAddress('host:port');
```

You can specify the host by name or by IPv4 address.

**Example app.js - daemon address**

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.setDaemonAddress('daemonhost:8082');
```

If you configured the daemon to listen on different ports for TCP and UDP, you can specify both in the daemon address setting.

**Example app.js - daemon address on separate ports**

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.setDaemonAddress('tcp:daemonhost:8082 udp:daemonhost:8083');
```

You can also set the daemon address by using the `AWS_XRAY_DAEMON_ADDRESS` environment variable (p. 219).

## Environment variables

You can use environment variables to configure the X-Ray SDK for Node.js. The SDK supports the following variables.

- `AWS_XRAY_CONTEXT_MISSING` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.

  **Valid Values**

  - `RUNTIME_ERROR` – Throw a runtime exception (default).
  - `LOG_ERROR` – Log an error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK uses `127.0.0.1:2000` for both trace data (UDP) and sampling (TCP). Use this variable if you have configured the daemon to listen on a different port (p. 140) or if it is running on a different host.

  **Format**

  - **Same port** – *`address`*`:`*`port`*
  - **Different ports** – `tcp:`*`address`*`:`*`port`* `udp:`*`address`*`:`*`port`*
- `AWS_XRAY_DEBUG_MODE` – Set to `TRUE` to configure the SDK to output logs to the console, instead of configuring a logger (p. 218).
- `AWS_XRAY_LOG_LEVEL` – Set a log level for the logger. Valid values are `debug`, `info`, `warn`, `error`, and `silent`. This value is ignored when AWS_XRAY_DEBUG_MODE is set to `TRUE`.
- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the segment name that you set on the Express middleware (p. 219).

# Tracing incoming requests with the X-Ray SDK for Node.js

You can use the X-Ray SDK for Node.js to trace incoming HTTP requests that your Express and Restify applications serve on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

The X-Ray SDK for Node.js provides middleware for applications that use the Express and Restify frameworks. When you add the X-Ray middleware to your application, the X-Ray SDK for Node.js creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

> **Note**
> For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray (p. 167) for more information.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The message handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

**Sections**



# Tracing incoming requests with Express

To use the Express middleware, initialize the SDK client and use the middleware returned by the `express.openSegment` function before you define your routes.

**Example app.js - Express**

```
var app = express();

var AWSXRay = require('aws-xray-sdk');
app.use(AWSXRay.express.openSegment('MyApp'));

app.get('/', function (req, res) {
  res.render('index');
});
```

```
app.use(AWSXRay.express.closeSegment());
```

After you define your routes, use the output of `express.closeSegment` as shown to handle any errors returned by the X-Ray SDK for Node.js.

# Tracing incoming requests with restify

To use the Restify middleware, initialize the SDK client and run `enable`. Pass it your Restify server and segment name.

**Example app.js - restify**

```
var AWSXRay = require('aws-xray-sdk');
var AWSXRayRestify = require('aws-xray-sdk-restify');

var restify = require('restify');
var server = restify.createServer();
AWSXRayRestify.enable(server, 'MyApp'));

server.get('/', function (req, res) {
  res.render('index');
});
```

# Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field (p. 103).

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you initialize the middleware, as shown in the previous sections.

> **Note**
> You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable (p. 219).

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request does not match the pattern. To name segments dynamically, use `AWSXRay.middleware.enableDynamicNaming`.

**Example app.js - dynamic segment names**

If the hostname in the request matches the pattern `*.example.com`, use the hostname. Otherwise, use `MyApp`.

```
var app = express();

var AWSXRay = require('aws-xray-sdk');
app.use(AWSXRay.express.openSegment('MyApp'));
AWSXRay.middleware.enableDynamicNaming('*.example.com');

app.get('/', function (req, res) {
  res.render('index');
});

app.use(AWSXRay.express.closeSegment());
```

# Tracing AWS SDK calls with the X-Ray SDK for Node.js

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Node.js tracks the calls downstream in subsegments (p. 225). Traced AWS services, and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the service map in the X-Ray console.

You can instrument all AWS SDK clients by wrapping your `aws-sdk` require statement in a call to `AWSXRay.captureAWS`.

**Example app.js - AWS SDK instrumentation**

```
var AWS = AWSXRay.captureAWS(require('aws-sdk'));
```

To instrument individual clients, wrap your AWS SDK client in a call to `AWSXRay.captureAWSClient`. For example, to instrument an `AmazonDynamoDB` client:

**Example app.js - DynamoDB client instrumentation**

```
    var AWSXRay = require('aws-xray-sdk');
...
    var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
```

> **Warning**
> Do not use both `captureAWS` and `captureAWSClient` together. This will lead to duplicate subsegments.

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

**Example Subsegment for a call to DynamoDB to save an item**

```
{
```

```
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

# Tracing calls to downstream HTTP web services using the X-Ray SDK for Node.js

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for Node.js client to instrument those calls and add the API to the service graph as a downstream service.

Pass your `http` or `https` client to the X-Ray SDK for Node.js `captureHTTPs` method to trace outgoing calls.

> **Note**
> Calls using third-party HTTP request libraries, such as Axios or Superagent, are supported through the `captureHTTPsGlobal()` API and will still be traced when they use the native `http` module.

**Example app.js - HTTP client**

```
var AWSXRay = require('aws-xray-sdk');
var http = AWSXRay.captureHTTPs(require('http'));
```

To enable tracing on all HTTP clients, call `captureHTTPsGlobal` before you load `http`.

**Example app.js - HTTP client (global)**

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.captureHTTPsGlobal(require('http'));
var http = require('http');
```

When you instrument a call to a downstream web API, the X-Ray SDK for Node.js records a subsegment that contains information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

**Example Subsegment for a downstream HTTP call**

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

**Example Inferred segment for a downstream HTTP call**

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-5880168b-fd5153bb58284b67678aa78c",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

# Tracing SQL queries with the X-Ray SDK for Node.js

Instrument SQL database queries by wrapping your SQL client in the corresponding X-Ray SDK for Node.js client method.

- **PostgreSQL** – `AWSXRay.capturePostgres()`

```
var AWSXRay = require('aws-xray-sdk');
var pg = AWSXRay.capturePostgres(require('pg'));
var client = new pg.Client();
```

- **MySQL** – `AWSXRay.captureMySQL()`

```
var AWSXRay = require('aws-xray-sdk');
```

```
var mysql = AWSXRay.captureMySQL(require('mysql'));
...
var connection = mysql.createConnection(config);
```

When you use an instrumented client to make SQL queries, the X-Ray SDK for Node.js records information about the connection and query in a subsegment.

## Including additional data in SQL subsegments

You can add additional information to subsegments generated for SQL queries, as long as it's mapped to a whitelisted SQL field. For example, to record the sanitized SQL query string in a subsegment, you can add it directly to the subsegment's SQL object.

**Example Assign SQL to sugsegment**

```
    const queryString = 'SELECT * FROM MyTable';
connection.query(queryString, ...);

// Retrieve the most recently created subsegment
const subs = AWSXRay.getSegment().subsegments;

if (subs & & subs.length > 0) {
  var sqlSub = subs[subs.length - 1];
  sqlSub.sql.sanitized_query = queryString;
}
```

For a full list of whitelisted SQL fields, see SQL Queries in the *AWS X-Ray Developer Guide*.

# Generating custom subsegments with the X-Ray SDK for Node.js

Subsegments extend a trace's segment (p. 21) with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

## Custom Express subsegments

To create a custom subsegment for a function that makes calls to downstream services, use the `captureAsyncFunc` function.

**Example app.js - custom subsegments Express**

```
var AWSXRay = require('aws-xray-sdk');

app.use(AWSXRay.express.openSegment('MyApp'));

app.get('/', function (req, res) {
  var host = 'api.example.com';

  AWSXRay.captureAsyncFunc('send', function(subsegment) {
```

```
    sendRequest(host, function() {
      console.log('rendering!');
      res.render('index');
      subsegment.close();
    });
  });
});

app.use(AWSXRay.express.closeSegment());

function sendRequest(host, cb) {
  var options = {
    host: host,
    path: '/',
  };

  var callback = function(response) {
    var str = '';

    response.on('data', function (chunk) {
      str += chunk;
    });

    response.on('end', function () {
      cb();
    });
  }

  http.request(options, callback).end();
};
```

In this example, the application creates a custom subsegment named `send` for calls to the `sendRequest` function. `captureAsyncFunc` passes a subsegment that you must close within the callback function when the asynchronous calls that it makes are complete.

For synchronous functions, you can use the `captureFunc` function, which closes the subsegment automatically as soon as the function block finishes executing.

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for Node.js generates an ID for it and records the start time and end time.

**Example Subsegment with metadata**

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

# Custom Lambda subsegments

The SDK is configured to automatically create a placeholder facade segment when the it detects it's running in Lambda. To create a basic subsegement, which will create a single `AWS::Lambda::Function` node on the X-Ray service map, call and repurpose the facade segment. If you manually create a new segment with a new ID (while sharing the trace ID, parent ID and the sampling decision) you will be able to send a new segment.

**Example app.js - manual custom subsegments**

```
const segment = AWSXRay.getSegment(); //returns the facade segment
const subsegment = segment.addNewSubsegment('subseg');
...
subsegment.close();
//the segment is closed by the SDK automatically
```

# Add annotations and metadata to segments with the X-Ray SDK for Node.js

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

**Annotations** are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions (p. 59). Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

**Metadata** are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings (p. 229) on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

**Sections**



## Recording annotations with the X-Ray SDK for Node.js

Use annotations to record information on segments or subsegments that you want indexed for search.

### Annotation Requirements

- **Keys** – Up to 500 alphanumeric characters. No spaces or symbols except underscores.
- **Values** – Up to 1,000 Unicode characters.
- **Entries** – Up to 50 annotations per trace.

### To record annotations

1. Get a reference to the current segment or subsegment.

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. Call `addAnnotation` with a String key, and a Boolean, Number, or String value.

```
document.addAnnotation("mykey", "my value");
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `addAnnotation` twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotations.key` keyword in a filter expression (p. 59).

**Example app.js - annotations**

```
var AWS = require('aws-sdk');
var AWSXRay = require('aws-xray-sdk');
var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
...
app.post('/signup', function(req, res) {
    var item = {
        'email': {'S': req.body.email},
        'name': {'S': req.body.name},
        'preview': {'S': req.body.previewAccess},
        'theme': {'S': req.body.theme}
    };

    var seg = AWSXRay.getSegment();
    seg.addAnnotation('theme', req.body.theme);

    ddb.putItem({
      'TableName': ddbTable,
      'Item': item,
      'Expected': { email: { Exists: false } }
  }, function(err, data) {
...
```

# Recording metadata with the X-Ray SDK for Node.js

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any other object that can be serialized into a JSON object or array.

**To record metadata**

1. Get a reference to the current segment or subsegment.

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. Call `addMetadata` with a string key, a Boolean, number, string, or object value, and a string namespace.

```
document.addMetadata("my key", "my value", "my namespace");
```

or

Call `addMetadata` with just a key and value.

```
document.addMetadata("my key", "my value");
```

If you don't specify a namespace, the SDK uses `default`. Calling `addMetadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

# Recording user IDs with the X-Ray SDK for Node.js

Record user IDs on request segments to identify the user who sent the request. This operation isn't compatible with AWS Lambda functions because segments in Lambda environments are immutable. The `setUser` call can be applied only to segments, not subsegments.

### To record user IDs

1. Get a reference to the current segment or subsegment.

```
var AWSXRay = require('aws-xray-sdk');
...
var document = AWSXRay.getSegment();
```

2. Call `setUser()` with a string ID of the user who sent the request.

```
var user = 'john123';

AWSXRay.getSegment().setUser(user);
```

You can call `setUser` to record the user ID as soon as your express application starts processing a request. If you will use the segment only to set the user ID, you can chain the calls in a single line.

### Example app.js - user ID

```
var AWS = require('aws-sdk');
var AWSXRay = require('aws-xray-sdk');
var uuidv4 = require('uuid/v4');
var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
...
    app.post('/signup', function(req, res) {
    var userId = uuidv4();
    var item = {
        'userId': {'S': userId},
        'email': {'S': req.body.email},
        'name': {'S': req.body.name}
    };

    var seg = AWSXRay.getSegment().setUser(userId);

    ddb.putItem({
      'TableName': ddbTable,
      'Item': item,
      'Expected': { email: { Exists: false } }
  }, function(err, data) {
...
```

To find traces for a user ID, use the `user` keyword in a filter expression.

# AWS X-Ray SDK for Python

The X-Ray SDK for Python is a library for Python web applications that provides classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an SQL database connector. You can also create segments manually and add debug information in annotations and metadata.

You can download the SDK with `pip`.

```
$ pip install aws-xray-sdk
```

> **Note**
> The X-Ray SDK for Python is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-python

If you use Django or Flask, start by adding the SDK middleware to your application (p. 237) to trace incoming requests. The middleware creates a segment (p. 21) for each traced request, and completes the segment when the response is sent. While the segment is open, you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open. For other applications, you can create segments manually (p. 239).

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header (p. 26) and traces sampled requests automatically. For other functions, you can configure Lambda (p. 167) to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

> **Note**
> On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray (p. 167) for more information.

See Worker (p. 125) for a example Python function instrumented in Lambda.

Next, use the X-Ray SDK for Python to instrument downstream calls by patching your application's libraries (p. 240). The SDK supports the following libraries.

### Supported Libraries

- `botocore`, `boto3` – Instrument AWS SDK for Python (Boto) clients.
- `pynamodb` – Instrument PynamoDB's version of the Amazon DynamoDB client.
- `aiobotocore`, `aioboto3` – Instrument asyncio-integrated versions of SDK for Python clients.
- `requests`, `aiohttp` – Instrument high-level HTTP clients.
- `httplib`, `http.client` – Instrument low-level HTTP clients and the higher level libraries that use them.
- `sqlite3` – Instrument SQLite clients.
- `mysql-connector-python` – Instrument MySQL clients.

- `pymysql` – Instrument PyMySQL based clients for MySQL and MariaDB.

Whenever your application makes calls to AWS, an SQL database, or other HTTP services, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the service map to help you identify errors and throttling issues on individual connections.

Once you get going with the SDK, customize its behavior by configuring the recorder and middleware (p. 232). You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata (p. 245). Annotations are simple key-value pairs that are indexed for use with filter expressions (p. 59), so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

> **Annotations and Metadata**
> Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments (p. 244). You can create a custom subsegment for an entire function or any section of code. You can then you can record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation for the SDK's classes and methods, see the AWS X-Ray SDK for Python API Reference.

# Requirements

The X-Ray SDK for Python supports the following language and library versions.

- **Python** – 2.7, 3.4, and newer
- **Django** – 1.10 and newer
- **Flask** – 0.10 and newer
- **aiohttp** – 2.3.0 and newer
- **AWS SDK for Python (Boto)** – 1.4.0 and newer
- **botocore** – 1.5.0 and newer
- **enum** – 0.4.7 and newer, for Python versions 3.4.0 and older
- **jsonpickle** – 1.0.0 and newer
- **setuptools** – 40.6.3 and newer
- **wrapt** – 1.11.0 and newer

# Dependency management

The X-Ray SDK for Python is available from `pip`.

- **Package** – `aws-xray-sdk`

Add the SDK as a dependency in your `requirements.txt` file.

**Example requirements.txt**

```
aws-xray-sdk==2.4.2
boto3==1.4.4
botocore==1.5.55
Django==1.11.3
```

If you use Elastic Beanstalk to deploy your application, Elastic Beanstalk installs all of the packages in `requirements.txt` automatically.

# Configuring the X-Ray SDK for Python

The X-Ray SDK for Python has a class named `xray_recorder` that provides the global recorder. You can configure the global recorder to customize the middleware that creates segments for incoming HTTP calls.

**Sections**

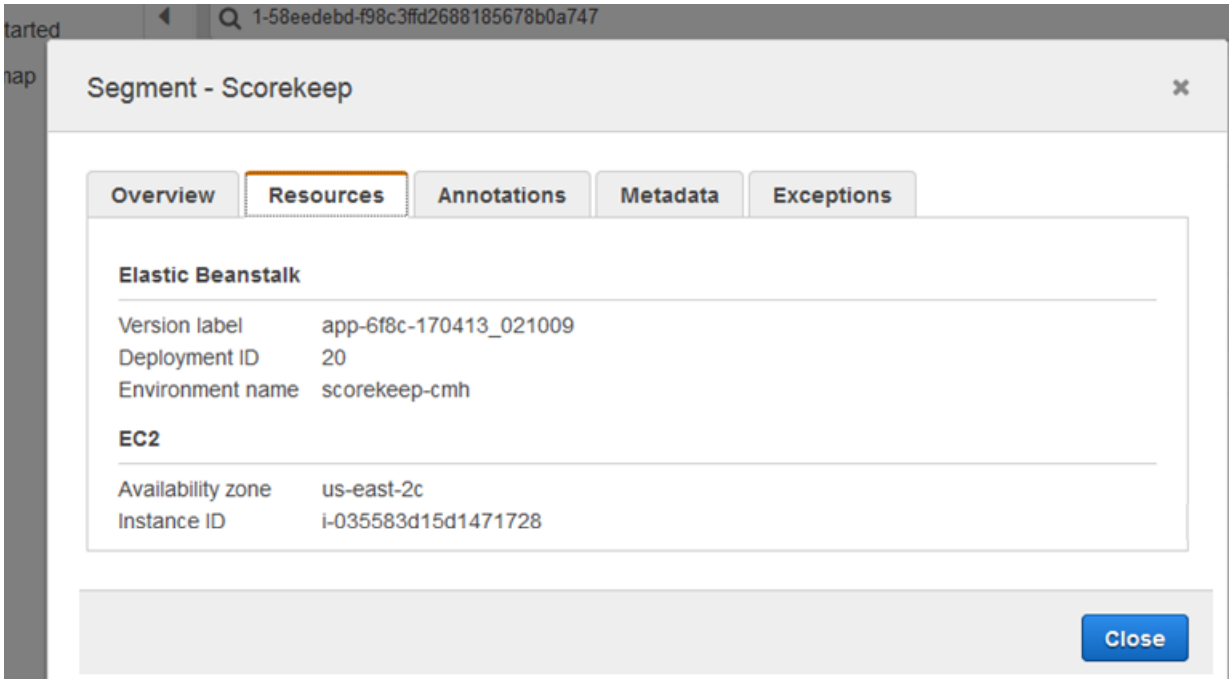


## Service plugins

Use `plugins` to record information about the service hosting your application.

**Plugins**

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.



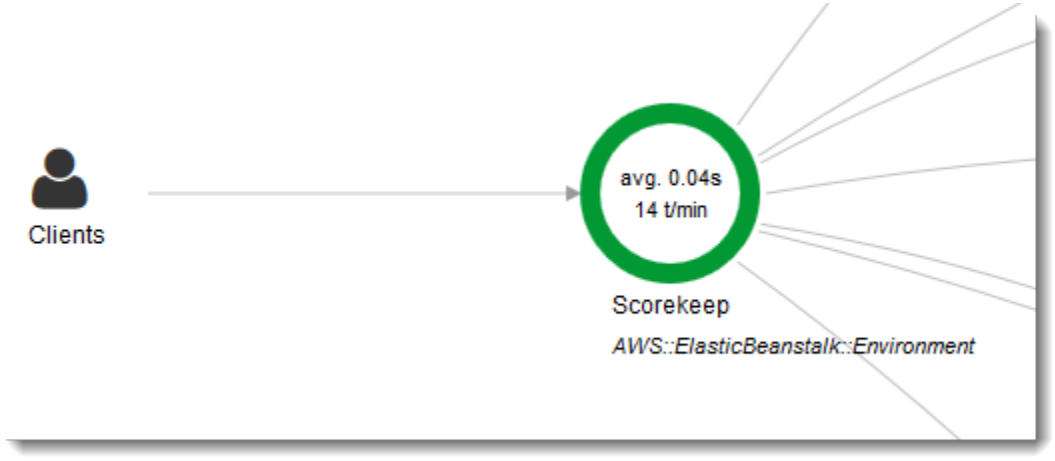

To use a plugin, call `configure` on the `xray_recorder`.

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

xray_recorder.configure(service='My app')
plugins = ('ElasticBeanstalkPlugin', 'EC2Plugin')
xray_recorder.configure(plugins=plugins)
patch_all()
```

You can also use environment variables (p. 236), which take precedence over values set in code, to configure the recorder.

Configure plugins before patching libraries (p. 232) to record downstream calls.

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. The resource type appears under your application's name in the service map. For example, `AWS::ElasticBeanstalk::Environment`.



When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

# Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console (p. 70) to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

**Example sampling-rules.json**

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To configure backup sampling rules, call `xray_recorder.configure`, as shown in the following example, where *`rules`* is either a dictionary of rules or the absolute path to a JSON file containing sampling rules.

```
xray_recorder.configure(sampling_rules=rules)
```

To use only local rules, configure the recorder with a `LocalSampler`.

```
from aws_xray_sdk.core.sampling.local.sampler import LocalSampler
xray_recorder.configure(sampler=LocalSampler())
```

You can also configure the global recorder to disable sampling and instrument all incoming requests.

**Example main.py – Disable sampling**

```
xray_recorder.configure(sampling=False)
```

# Logging

The SDK uses Python's built-in `logging` module. Get a reference to the logger for the `aws_xray_sdk` class and call `setLevel` on it to configure the different log level for the library and the rest of your application.

**Example app.py – Logging**

```
logging.basicConfig(level='WARNING')
logging.getLogger('aws_xray_sdk').setLevel(logging.DEBUG)
```

Use debug logs to identify issues, such as unclosed subsegments, when you generate subsegments manually (p. 244).

# Recorder configuration in code

Additional settings are available from the `configure` method on `xray_recorder`.

- `context_missing` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.
- `daemon_address` – Set the host and port of the X-Ray daemon listener.
- `service` – Set a service name that the SDK uses for segments.
- `plugins` – Record information about your application's AWS resources.
- `sampling` – Set to `False` to disable sampling.
- `sampling_rules` – Set the path of the JSON file containing your sampling rules (p. 234).

**Example main.py – Disable context missing exceptions**

```
from aws_xray_sdk.core import xray_recorder

xray_recorder.configure(context_missing='LOG_ERROR')
```

# Recorder configuration with Django

If you use the Django framework, you can use the Django `settings.py` file to configure options on the global recorder.

- `AUTO_INSTRUMENT` (Django only) – Record subsegments for built-in database and template rendering operations.
- `AWS_XRAY_CONTEXT_MISSING` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener.

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments.
- `PLUGINS` – Record information about your application's AWS resources.
- `SAMPLING` – Set to `False` to disable sampling.
- `SAMPLING_RULES` – Set the path of the JSON file containing your sampling rules (p. 234).

To enable recorder configuration in `settings.py`, add the Django middleware to the list of installed apps.

**Example settings.py – Installed apps**

```
INSTALLED_APPS = [
    ...
    'django.contrib.sessions',
    'aws_xray_sdk.ext.django',
]
```

Configure the available settings in a dict named `XRAY_RECORDER`.

**Example settings.py – Installed apps**

```
XRAY_RECORDER = {
    'AUTO_INSTRUMENT': True,
    'AWS_XRAY_CONTEXT_MISSING': 'LOG_ERROR',
    'AWS_XRAY_DAEMON_ADDRESS': '127.0.0.1:5000',
    'AWS_XRAY_TRACING_NAME': 'My application',
    'PLUGINS': ('ElasticBeanstalkPlugin', 'EC2Plugin', 'ECSPlugin'),
    'SAMPLING': False,
}
```

# Environment variables

You can use environment variables to configure the X-Ray SDK for Python. The SDK supports the following variables:

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the service name that you set programmatically.
- `AWS_XRAY_SDK_ENABLED` – When set to `false`, disables the SDK. By default, the SDK is enabled unless the environment variable is set to false.
  - When disabled, the global recorder automatically generates dummy segments and subsegments that are not sent to the daemon, and automatic patching is disabled. Middlewares are written as a wrapper over the global recorder. All segment and subsegment generation through the middleware also become dummy segment and dummy subsegments.
  - Set the value of `AWS_XRAY_SDK_ENABLED` through the environment variable or through direct interaction with the `global_sdk_config` object from the `aws_xray_sdk` library. Settings to the environment variable override these interactions.
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK uses `127.0.0.1:2000` for both trace data (UDP) and sampling (TCP). Use this variable if you have configured the daemon to listen on a different port (p. 140) or if it is running on a different host.

  **Format**

  - **Same port** – *`address`*`:`*`port`*
  - **Different ports** – `tcp:`*`address`*`:`*`port`* `udp:`*`address`*`:`*`port`*
- `AWS_XRAY_CONTEXT_MISSING` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.

**Valid Values**

- `RUNTIME_ERROR` – Throw a runtime exception (default).
- `LOG_ERROR` – Log an error and continue.

Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

Environment variables override values set in code.

# Tracing incoming requests with the X-Ray SDK for Python middleware

When you add the middleware to your application and configure a segment name, the X-Ray SDK for Python creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

When you add the middleware to your application and configure a segment name, the X-Ray SDK for Python creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

The X-Ray SDK for Python supports the following middleware to instrument incoming HTTP requests:

- Django
- Flask
- Bottle

**Note**
For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray (p. 167) for more information.

See Worker (p. 125) for a example Python function instrumented in Lambda.

For scripts or Python applications on other frameworks, you can create segments manually (p. 239).

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

**Forwarded Requests**
If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The middleware creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

**Sections**



# Adding the middleware to your application (Django)

Add the middleware to the `MIDDLEWARE` list in your `settings.py` file. The X-Ray middleware should be the first line in your `settings.py` file to ensure that requests that fail in other middleware are recorded.

**Example settings.py - X-Ray SDK for Python middleware**

```
MIDDLEWARE = [
    'aws_xray_sdk.ext.django.middleware.XRayMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware'
]
```

Configure a segment name in your `settings.py` file (p. 235).

**Example settings.py – Segment name**

```
XRAY_RECORDER = {,
    'AWS_XRAY_TRACING_NAME': 'My application',
    'PLUGINS': ('EC2Plugin'),
}
```

This tells the X-Ray recorder to trace requests served by your Django application with the default sampling rate. You can configure the recorder your Django settings file (p. 235) to apply custom sampling rules or change other settings.

# Adding the middleware to your application (flask)

To instrument your Flask application, first configure a segment name on the `xray_recorder`. Then, use the `XRayMiddleware` function to patch your Flask application in code.

**Example app.py**

```
from aws_xray_sdk.core import xray_recorder
```

```
from aws_xray_sdk.ext.flask.middleware import XRayMiddleware

app = Flask(__name__)

xray_recorder.configure(service='My application')
XRayMiddleware(app, xray_recorder)
```

This tells the X-Ray recorder to trace requests served by your Flask application with the default sampling rate. You can configure the recorder in code (p. 235) to apply custom sampling rules or change other settings.

# Adding the middleware to your application (Bottle)

To instrument your Bottle application, first configure a segment name on the `xray_recorder`. Then, use the `XRayMiddleware` function to patch your Bottle application in code.

**Example app.py**

```
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.ext.bottle.middleware import XRayMiddleware

app = Bottle()

xray_recorder.configure(service='fallback_name', dynamic_naming='My application')
app.install(XRayMiddleware(xray_recorder))
```

This tells the X-Ray recorder to trace requests served by your Bottle application with the default sampling rate. You can configure the recorder in code (p. 235) to apply custom sampling rules or change other settings.

# Instrumenting Python code manually

If you don't use Django or Flask, you can create segments manually. You can create a segment for each incoming request, or create segments around patched HTTP or AWS SDK clients to provide context for the recorder to add subsegments.

**Example main.py – Manual instrumentation**

```
from aws_xray_sdk.core import xray_recorder

# Start a segment
segment = xray_recorder.begin_segment('segment_name')
# Start a subsegment
subsegment = xray_recorder.begin_subsegment('subsegment_name')

# Add metadata and annotations
segment.put_metadata('key', dict, 'namespace')
subsegment.put_annotation('key', 'value')

# Close the subsegment and segment
xray_recorder.end_subsegment()
xray_recorder.end_segment()
```

# Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates

segments for incoming requests, it records your application's service name in the segment's name field (p. 103).

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you configure the recorder, as shown in the previous sections (p. 238).

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request doesn't match the pattern. To name segments dynamically in Django, add the `DYNAMIC_NAMING` setting to your settings.py (p. 235) file.

**Example settings.py – Dynamic naming**

```
XRAY_RECORDER = {
    'AUTO_INSTRUMENT': True,
    'AWS_XRAY_TRACING_NAME': 'My application',
    'DYNAMIC_NAMING': '*.example.com',
    'PLUGINS': ('ElasticBeanstalkPlugin', 'EC2Plugin')
}
```

You can use '*' in the pattern to match any string, or '?' to match any single character. For Flask, configure the recorder in code (p. 235).

**Example main.py – Segment name**

```
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='My application')
xray_recorder.configure(dynamic_naming='*.example.com')
```

> **Note**
> You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable (p. 236).

# Patching libraries to instrument downstream calls

To instrument downstream calls, use the X-Ray SDK for Python to patch the libraries that your application uses. The X-Ray SDK for Python can patch the following libraries.

**Supported Libraries**

- `botocore`, `boto3` – Instrument AWS SDK for Python (Boto) clients.

- `pynamodb` – Instrument PynamoDB's version of the Amazon DynamoDB client.
- `aiobotocore`, `aioboto3` – Instrument asyncio-integrated versions of SDK for Python clients.
- `requests`, `aiohttp` – Instrument high-level HTTP clients.
- `httplib`, `http.client` – Instrument low-level HTTP clients and the higher level libraries that use them.
- `sqlite3` – Instrument SQLite clients.
- `mysql-connector-python` – Instrument MySQL clients.
- `pymysql` – Instrument PyMySQL based clients for MySQL and MariaDB.

When you use a patched library, the X-Ray SDK for Python creates a subsegment for the call and records information from the request and response. A segment must be available for the SDK to create the subsegment, either from the SDK middleware or from AWS Lambda.

> **Note**
> If you use SQLAlchemy ORM, you can instrument your SQL queries by importing the SDK's version of SQLAlchemy's session and query classes. See Use SQLAlchemy ORM for instructions.

To patch all available libraries, use the `patch_all` function in `aws_xray_sdk.core`. Some libraries, such as `httplib` and `urllib`, may need to enable double patching by calling `patch_all(double_patch=True)`.

**Example main.py – Patch all supported libraries**

```
import boto3
import botocore
import requests
import sqlite3

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

patch_all()
```

To patch a single library, call `patch` with a tuple of the library name. In order to achieve this, you will need to provide a single element list.

**Example main.py – Patch specific libraries**

```
import boto3
import botocore
import requests
import mysql-connector-python

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch

libraries = (['botocore'])
patch(libraries)
```

> **Note**
> In some cases, the key that you use to patch a library does not match the library name. Some keys serve as aliases for one or more libraries.
>
> **Libraries Aliases**
>
> - `httplib` – `httplib` and `http.client`

- `mysql` – `mysql-connector-python`

## Tracing context for asynchronous work

For `asyncio` integrated libraries, or to create subsegments for asynchronous functions (p. 244), you must also configure the X-Ray SDK for Python with an async context. Import the `AsyncContext` class and pass an instance of it to the X-Ray recorder.

> **Note**
> Web framework support libraries, such as AIOHTTP, are not handled through the `aws_xray_sdk.core.patcher` module. They will not appear in the `patcher` catalog of supported libraries.

**Example main.py – Patch aioboto3**

```
import asyncio
import aioboto3
import requests

from aws_xray_sdk.core.async_context import AsyncContext
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='my_service', context=AsyncContext())
from aws_xray_sdk.core import patch

libraries = ('aioboto3')
patch(libraries)
```

# Tracing AWS SDK calls with the X-Ray SDK for Python

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Python tracks the calls downstream in subsegments (p. 244). Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the service map in the X-Ray console.

The X-Ray SDK for Python automatically instruments all AWS SDK clients when you patch the `botocore` library (p. 240). You cannot instrument individual clients.

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

**Example Subsegment for a call to DynamoDB to save an item**

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
```

```
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

# Tracing calls to downstream HTTP web services using the X-Ray SDK for Python

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for Python to instrument those calls and add the API to the service graph as a downstream service.

To instrument HTTP clients, patch the library (p. 240) that you use to make outgoing calls. If you use `requests` or Python's built in HTTP client, that's all you need to do. For `aiohttp`, also configure the recorder with an async context (p. 242).

If you use `aiohttp` 3's client API, you also need to configure the `ClientSession`'s with an instance of the tracing configuration provided by the SDK.

**Example `aiohttp` 3 client API**

```
from aws_xray_sdk.ext.aiohttp.client import aws_xray_trace_config

async def foo():
    trace_config = aws_xray_trace_config()
    async with ClientSession(loop=loop, trace_configs=[trace_config]) as session:
        async with session.get(url) as resp
            await resp.read()
```

When you instrument a call to a downstream web API, the X-Ray SDK for Python records a subsegment that contains information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the remote API.

**Example Subsegment for a downstream HTTP call**

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
  "http": {
    "request": {
      "method": "GET",
```

```
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

**Example Inferred segment for a downstream HTTP call**

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-5880168b-fd5153bb58284b67678aa78c",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

# Generating custom subsegments with the X-Ray SDK for Python

Subsegments extend a trace's segment (p. 21) with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

To manage subsegments, use the `begin_subsegment` and `end_subsegment` methods.

**Example main.py – Custom subsegment**

```
from aws_xray_sdk.core import xray_recorder

subsegment = xray_recorder.begin_subsegment('annotations')
subsegment.put_annotation('id', 12345)
xray_recorder.end_subsegment()
```

To create a subsegment for a synchronous function, use the `@xray_recorder.capture` decorator. You can pass a name for the subsegment to the capture function or leave it out to use the function name.

**Example main.py – Function subsegment**

```
from aws_xray_sdk.core import xray_recorder
```

```
@xray_recorder.capture('## create_user')
def create_user():
...
```

For an asynchronous function, use the `@xray_recorder.capture_async` decorator, and pass an async context to the recorder.

**Example main.py – Asynchronous function subsegment**

```
from aws_xray_sdk.core.async_context import AsyncContext
from aws_xray_sdk.core import xray_recorder
xray_recorder.configure(service='my_service', context=AsyncContext())

@xray_recorder.capture_async('## create_user')
async def create_user():
    ...

async def main():
    await myfunc()
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for Python generates an ID for it and records the start time and end time.

**Example Subsegment with metadata**

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

# Add annotations and metadata to segments with the X-Ray SDK for Python

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

**Annotations** are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions (p. 59). Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

**Metadata** are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings (p. 247) on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

**Sections**

# Recording annotations with the X-Ray SDK for Python

Use annotations to record information on segments or subsegments that you want indexed for search.

### Annotation Requirements

- **Keys** – Up to 500 alphanumeric characters. No spaces or symbols except underscores.
- **Values** – Up to 1,000 Unicode characters.
- **Entries** – Up to 50 annotations per trace.

### To record annotations

1. Get a reference to the current segment or subsegment from `xray_recorder`.

   ```
   from aws_xray_sdk.core import xray_recorder
   ...
   document = xray_recorder.current_segment()
   ```

   or

   ```
   from aws_xray_sdk.core import xray_recorder
   ...
   document = xray_recorder.current_subsegment()
   ```

2. Call `put_annotation` with a String key, and a Boolean, Number, or String value.

   ```
   document.put_annotation("mykey", "my value");
   ```

Alternatively, you can use the `put_annotation` method on the `xray_recorder`. This method records annotations on the current subsegment or, if no subsegment is open, on the segment.

```
xray_recorder.put_annotation("mykey", "my value");
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `put_annotation` twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotations.`*`key`* keyword in a filter expression (p. 59).

# Recording metadata with the X-Ray SDK for Python

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any object that can be serialized into a JSON object or array.

**To record metadata**

1. Get a reference to the current segment or subsegment from `xray_recorder`.

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_segment()
```

or

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_subsegment()
```

2. Call `put_metadata` with a String key; a Boolean, Number, String, or Object value; and a String namespace.

```
document.put_metadata("my key", "my value", "my namespace");
```

or

Call `put_metadata` with just a key and value.

```
document.put_metadata("my key", "my value");
```

Alternatively, you can use the `put_metadata` method on the `xray_recorder`. This method records metadata on the current subsegment or, if no subsegment is open, on the segment.

```
xray_recorder.put_metadata("my key", "my value");
```

If you don't specify a namespace, the SDK uses `default`. Calling `put_metadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

# Recording user IDs with the X-Ray SDK for Python

Record user IDs on request segments to identify the user who sent the request.

**To record user IDs**

1. Get a reference to the current segment from `xray_recorder`.

```
from aws_xray_sdk.core import xray_recorder
...
document = xray_recorder.current_segment()
```

2. Call `setUser` with a String ID of the user who sent the request.

```
document.set_user("U12345");
```

You can call `set_user` in your controllers to record the user ID as soon as your application starts processing a request.

To find traces for a user ID, use the `user` keyword in a filter expression (p. 59).

# Instrumenting web frameworks deployed to serverless environments

The AWS X-Ray SDK for Python supports instrumenting web frameworks deployed in serverless applications. Serverless is the native architecture of the cloud that enables you to shift more of your operational responsibilities to AWS, increasing your agility and innovation.

Serverless architecture is a software application model that enables you to build and run applications and services without thinking about servers. It eliminates infrastructure management tasks such as server or cluster provisioning, patching, operating system maintenance, and capacity provisioning. You can build serverless solutions for nearly any type of application or backend service, and everything required to run and scale your application with high availability is handled for you.

This tutorial shows you how to automatically instrument AWS X-Ray on a web framework, such as Flask or Django, that is deployed to a serverless environment. X-Ray instrumentation of the application enables you to view all downstream calls that are made, starting from Amazon API Gateway through your AWS Lambda function, and the outgoing calls your application makes.

The X-Ray SDK for Python supports the following Python application frameworks:

- Flask version 0.8, or later
- Django version 1.0, or later

This tutorial develops an example serverless application that is deployed to Lambda and invoked by API Gateway. This tutorial uses Zappa to automatically deploy the application to Lambda and to configure the API Gateway endpoint.

## Prerequisites

- Zappa
- Python – Version 2.7 or 3.6.
- AWS CLI – Verify that your AWS CLI is configured with the account and AWS Region in which you will deploy your application.
- Pip
- Virtualenv

## Step 1: Create an environment

In this step, you create a virtual environment using `virtualenv` to host an application.

1. Using the AWS CLI, create a directory for the application. Then change to the new directory.

```
mkdir serverless_application
cd serverless_application
```

2. Next, create a virtual environment within your new directory. Use the following command to activate it.

```
# Create our virtual environment
virtualenv serverless_env
```

```
# Activate it
source serverless_env/bin/activate
```

3. Install X-Ray, Flask, Zappa, and the Requests library to your environment.

```
# Install X-Ray, Flask, Zappa, and Requests into your environment
pip install aws-xray-sdk flask zappa requests
```

4. Add application code to the `serverless_application` directory. For this example, we can build off of Flasks's Hello World example.

   In the `serverless_application` directory, create a file named `my_app.py`. Then use a text editor to add the following commands. This application instruments the Requests library, patches the Flask application's middleware, and opens the endpoint `'/'`.

```
# Import the X-Ray modules
from aws_xray_sdk.ext.flask.middleware import XRayMiddleware
from aws_xray_sdk.core import patcher, xray_recorder
from flask import Flask
import requests

# Patch the requests module to enable automatic instrumentation
patcher.patch(('requests',))

app = Flask(__name__)

# Configure the X-Ray recorder to generate segments with our service name
xray_recorder.configure(service='My First Serverless App')

# Instrument the Flask application
XRayMiddleware(app, xray_recorder)

@app.route('/')
def hello_world():
    resp = requests.get("https://aws.amazon.com")
    return 'Hello, World: %s' % resp.url
```

# Step 2: Create and deploy a zappa environment

In this step you will use Zappa to automatically configure an API Gateway endpoint and then deploy to Lambda.

1. Initialize Zappa from within the `serverless_application` directory. For this example, we used the default settings, but if you have customization preferences, Zappa displays configuration instructions.

```
zappa init
```

```
What do you want to call this environment (default 'dev'): dev
...
What do you want to call your bucket? (default 'zappa-*******'): zappa-*******
...
...
It looks like this is a Flask application.
What's the modular path to your app's function?
This will likely be something like 'your_module.app'.
We discovered: my_app.app
Where is your app's function? (default 'my_app.app'): my_app.app
```

```
...
Would you like to deploy this application globally? (default 'n') [y/n/(p)rimary]: n
```

2. Enable X-Ray. Open the `zappa_settings.json` file and verify that it looks similar to the example.

```
{
    "dev": {
        "app_function": "my_app.app",
        "aws_region": "us-west-2",
        "profile_name": "default",
        "project_name": "serverless-exam",
        "runtime": "python2.7",
        "s3_bucket": "zappa-*********"
    }
}
```

3. Add `"xray_tracing": true` as an entry to the configuration file.

```
{
    "dev": {
        "app_function": "my_app.app",
        "aws_region": "us-west-2",
        "profile_name": "default",
        "project_name": "serverless-exam",
        "runtime": "python2.7",
        "s3_bucket": "zappa-*********",
        "xray_tracing": true
    }
}
```

4. Deploy the application. This automatically configures the API Gateway endpoint and uploads your code to Lambda.

```
zappa deploy
```

```
...
Deploying API Gateway..
Deployment complete!: https://**********.execute-api.us-west-2.amazonaws.com/dev
```

# Step 3: Enable X-Ray tracing for API Gateway

In this step you will interact with the API Gateway console to enable X-Ray tracing.

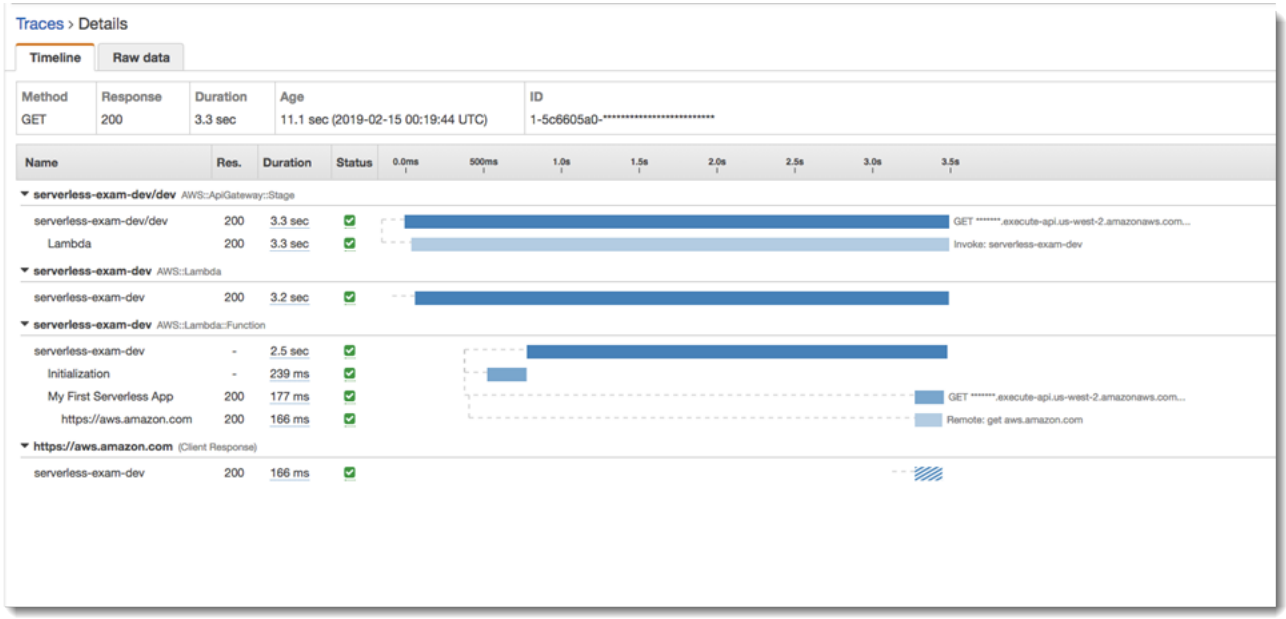
1. Sign in to the AWS Management Console and open the API Gateway console at https://console.aws.amazon.com/apigateway/.
2. Find your newly generated API. It should look something like `serverless-exam-dev`.
3. Choose **Stages**.
4. Choose the name of your deployment stage. The default is `dev`.
5. On the **Logs/Tracing** tab, select the **Enable X-Ray Tracing** box.
6. Choose **Save Changes**.
7. Access the endpoint in your browser. If you used the example `Hello World` application, it should display the following.

```
"Hello, World: https://aws.amazon.com/"
```

# Step 4: View the created trace

In this step you will interact with the X-Ray console to view the trace created by the example application. For a more detailed walkthrough on trace analysis, see Viewing the Service Map.

1. Sign in to the AWS Management Console and open the X-Ray console at https://console.aws.amazon.com/xray/home.
2. View segments generated by API Gateway, the Lambda function, and the Lambda container.
3. Under the Lambda function segment, view a subsegment named `My First Serverless App`. It's followed by a second subsegment named `https://aws.amazon.com`.
4. During initialization, Lambda might also generate a third subsegment named `initialization`.




# Step 5: Clean up

Always terminate resources you are no longer using to avoid the accumulation of unexpected costs. As this tutorial demonstrates, tools such as Zappa streamline serverless redeployment.

To remove your application from Lambda, API Gateway, and Amazon S3, run the following command in your project directory by using the AWS CLI.

```
zappa undeploy dev
```

# Next steps

Add more features to your application by adding AWS clients and instrumenting them with X-Ray. Learn more about serverless computing options through AWS at Serverless.

# AWS X-Ray SDK for Ruby

The X-Ray SDK is a library for Ruby web applications that provides classes and methods for generating and sending trace data to the X-Ray daemon. Trace data includes information about incoming HTTP requests served by the application, and calls that the application makes to downstream services using the AWS SDK, HTTP clients, or an active record client. You can also create segments manually and add debug information in annotations and metadata.

You can download the SDK by adding it to your gemfile and running `bundle install`.

**Example Gemfile**

```
gem 'aws-sdk-ruby'
```

If you use Rails, start by adding the X-Ray SDK middleware (p. 259) to trace incoming requests. A request filter creates a segment (p. 21). While the segment is open, you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open. For non-Rails applications, you can create segments manually (p. 260).

Next, use the X-Ray SDK to instrument your AWS SDK for Ruby, HTTP, and SQL clients by configuring the recorder (p. 261) to patch the associated libraries. Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the service map to help you identify errors and throttling issues on individual connections.

Once you get going with the SDK, customize its behavior by configuring the recorder (p. 254). You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and provide a logger to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata (p. 263). Annotations are simple key-value pairs that are indexed for use with filter expressions (p. 59), so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

> **Annotations and Metadata**
> Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have a lot of instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments (p. 262). You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation for the SDK's classes and methods, see the AWS X-Ray SDK for Ruby API Reference.

## Requirements

The X-Ray SDK requires Ruby 2.3 or later and is compatible with the following libraries:

- AWS SDK for Ruby version 3.0 or later

- Rails version 5.1 or later

# Configuring the X-Ray SDK for Ruby

The X-Ray SDK for Ruby has a class named `XRay.recorder` that provides the global recorder. You can configure the global recorder to customize the middleware that creates segments for incoming HTTP calls.

**Sections**

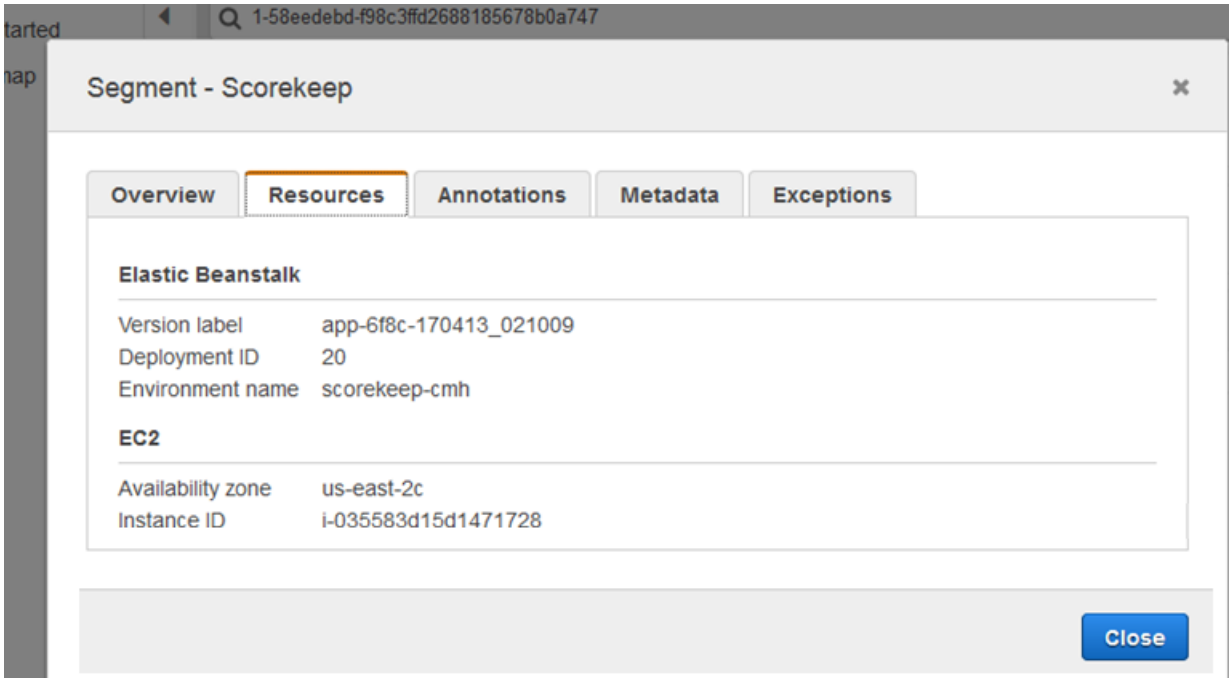


## Service plugins

Use `plugins` to record information about the service hosting your application.

**Plugins**

- Amazon EC2 – `ec2` adds the instance ID and Availability Zone.
- Elastic Beanstalk – `elastic_beanstalk` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ecs` adds the container ID.



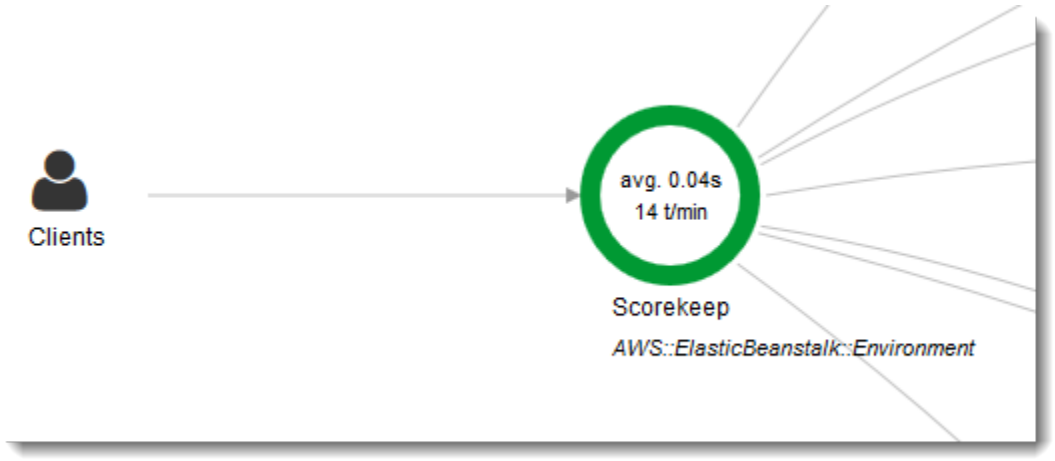

To use plugins, specify it in the configuration object that you pass to the recorder.

**Example main.rb – Plugin configuration**

```
my_plugins = %I[ec2 elastic_beanstalk]

config = {
  plugins: my_plugins,
  name: 'my app',
}

XRay.recorder.configure(config)
```

You can also use environment variables (p. 258), which take precedence over values set in code, to configure the recorder.

The SDK also uses plugin settings to set the `origin` field on the segment. This indicates the type of AWS resource that runs your application. The resource type appears under your application's name in the service map. For example, `AWS::ElasticBeanstalk::Environment`.




When you use multiple plugins, the SDK uses the following resolution order to determine the origin: ElasticBeanstalk > EKS > ECS > EC2.

# Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console (p. 70) to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

**Example sampling-rules.json**

```
{
```

```
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

To configure backup rules, define a hash for the document in the configuration object that you pass to the recorder.

**Example main.rb – Backup rule configuration**

```
require 'aws-xray-sdk'
my_sampling_rules =  {
  version: 1,
  default: {
    fixed_target: 1,
    rate: 0.1
  }
}
config = {
  sampling_rules: my_sampling_rules,
  name: 'my app',
}
XRay.recorder.configure(config)
```

To store the sampling rules independently, define the hash in a separate file and require the file to pull it into your application.

**Example config/sampling-rules.rb**

```
my_sampling_rules =  {
  version: 1,
  default: {
    fixed_target: 1,
    rate: 0.1
  }
}
```

**Example main.rb – Sampling rule from a file**

```
require 'aws-xray-sdk'
require config/sampling-rules.rb
```

```
config = {
  sampling_rules: my_sampling_rules,
  name: 'my app',
}
XRay.recorder.configure(config)
```

To use only local rules, require the sampling rules and configure the `LocalSampler`.

**Example main.rb – Local rule sampling**

```
require 'aws-xray-sdk'
require 'aws-xray-sdk/sampling/local/sampler'

config = {
  sampler: LocalSampler.new,
  name: 'my app',
}
XRay.recorder.configure(config)
```

You can also configure the global recorder to disable sampling and instrument all incoming requests.

**Example main.rb – Disable sampling**

```
require 'aws-xray-sdk'
config = {
  sampling: false,
  name: 'my app',
}
XRay.recorder.configure(config)
```

# Logging

By default, the recorder outputs info-level events to `$stdout`. You can customize logging by defining a logger in the configuration object that you pass to the recorder.

**Example main.rb – Logging**

```
require 'aws-xray-sdk'
config = {
  logger: my_logger,
  name: 'my app',
}
XRay.recorder.configure(config)
```

Use debug logs to identify issues, such as unclosed subsegments, when you generate subsegments manually (p. 262).

# Recorder configuration in code

Additional settings are available from the `configure` method on `XRay.recorder`.

- `context_missing` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.
- `daemon_address` – Set the host and port of the X-Ray daemon listener.
- `name` – Set a service name that the SDK uses for segments.
- `naming_pattern` – Set a domain name pattern to use dynamic naming (p. 260).

- `plugins` – Record information about your application's AWS resources with plugins (p. 254).
- `sampling` – Set to `false` to disable sampling.
- `sampling_rules` – Set the hash containing your sampling rules (p. 255).

**Example main.rb – Disable context missing exceptions**

```
require 'aws-xray-sdk'
config = {
  context_missing: 'LOG_ERROR'
}

XRay.recorder.configure(config)
```

# Recorder configuration with rails

If you use the Rails framework, you can configure options on the global recorder in a Ruby file under `app_root/initializers`. The X-Ray SDK supports an additional configuration key for use with Rails.

- `active_record` – Set to `true` to record subsegments for Active Record database transactions.

Configure the available settings in a configuration object named `Rails.application.config.xray`.

**Example config/initializers/aws_xray.rb**

```
Rails.application.config.xray = {
  name: 'my app',
  patch: %I[net_http aws_sdk],
  active_record: true
}
```

# Environment variables

You can use environment variables to configure the X-Ray SDK for Ruby. The SDK supports the following variables:

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the service name that you set on the servlet filter's segment naming strategy (p. 260).
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK sends trace data to `127.0.0.1:2000`. Use this variable if you have configured the daemon to listen on a different port (p. 140) or if it is running on a different host.
- `AWS_XRAY_CONTEXT_MISSING` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.

  **Valid Values**

  - `RUNTIME_ERROR` – Throw a runtime exception (default).
  - `LOG_ERROR` – Log an error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

Environment variables override values set in code.

# Tracing incoming requests with the X-Ray SDK for Ruby middleware

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

If you use Rails, use the Rails middleware to instrument incoming HTTP requests. When you add the middleware to your application and configure a segment name, the X-Ray SDK for Ruby creates a segment for each sampled request. Any segments created by additional instrumentation become subsegments of the request-level segment that provides information about the HTTP request and response. This information includes timing, method, and disposition of the request.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

When a request is forwarded, the SDK sets an additional field in the segment to indicate this. If the segment contains the field `x_forwarded_for` set to `true`, the client IP was taken from the `X-Forwarded-For` header in the HTTP request.

The middleware creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

## Using the rails middleware

To use the middleware, update your gemfile to include the required railtie.

**Example Gemfile - rails**

```
gem 'aws-xray-sdk', require: ['aws-xray-sdk/facets/rails/railtie']
```

To use the middleware, you must also configure the recorder (p. 258) with a name that represents the application in the service map.

**Example config/initializers/aws_xray.rb**

```
Rails.application.config.xray = {
  name: 'my app'
}
```

# Instrumenting code manually

If you don't use Rails, create segments manually. You can create a segment for each incoming request, or create segments around patched HTTP or AWS SDK clients to provide context for the recorder to add subsegments.

```
# Start a segment
segment = XRay.recorder.begin_segment 'my_service'
# Start a subsegment
subsegment = XRay.recorder.begin_subsegment 'outbound_call', namespace: 'remote'

# Add metadata or annotation here if necessary
my_annotations = {
  k1: 'v1',
  k2: 1024
}
segment.annotations.update my_annotations

# Add metadata to default namespace
subsegment.metadata[:k1] = 'v1'

# Set user for the segment (subsegment is not supported)
segment.user = 'my_name'

# End segment/subsegment
XRay.recorder.end_subsegment
XRay.recorder.end_segment
```

# Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field (p. 103).

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you configure the recorder, as shown in the previous sections (p. 259).

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request doesn't match the pattern. To name segments dynamically, specify a naming pattern in the config hash.

**Example main.rb – Dynamic naming**

```
config = {
  naming_pattern: '*mydomain*',
  name: 'my app',
}

XRay.recorder.configure(config)
```

You can use '*' in the pattern to match any string, or '?' to match any single character.

> **Note**
> You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable (p. 258).

# Patching libraries to instrument downstream calls

To instrument downstream calls, use the X-Ray SDK for Ruby to patch the libraries that your application uses. The X-Ray SDK for Ruby can patch the following libraries.

**Supported Libraries**

- `net/http` – Instrument HTTP clients.
- `aws-sdk` – Instrument AWS SDK for Ruby clients.

When you use a patched library, the X-Ray SDK for Ruby creates a subsegment for the call and records information from the request and response. A segment must be available for the SDK to create the subsegment, either from the SDK middleware or a call to `XRay.recorder.begin_segment`.

To patch libraries, specify them in the configuration object that you pass to the X-Ray recorder.

**Example main.rb – Patch libraries**

```
require 'aws-xray-sdk'

config = {
  name: 'my app',
  patch: %I[net_http aws_sdk]
}

XRay.recorder.configure(config)
```

# Tracing AWS SDK calls with the X-Ray SDK for Ruby

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for Ruby tracks the calls downstream in subsegments (p. 262). Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the service map in the X-Ray console.

The X-Ray SDK for Ruby automatically instruments all AWS SDK clients when you patch the `aws-sdk` library (p. 261). You cannot instrument individual clients.

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

**Example Subsegment for a call to DynamoDB to save an item**

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name
- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

# Generating custom subsegments with the X-Ray SDK

Subsegments extend a trace's segment (p. 21) with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

To manage subsegments, use the `begin_subsegment` and `end_subsegment` methods.

```
subsegment = XRay.recorder.begin_subsegment name: 'annotations', namespace: 'remote'
my_annotations = { id: 12345 }
subsegment.annotations.update my_annotations
XRay.recorder.end_subsegment
```

To create a subsegment for a function, wrap it in a call to `XRay.recorder.capture`.

```
XRay.recorder.capture('name_for_subsegment') do |subsegment|
  resp = myfunc() # myfunc is your function
  subsegment.annotations.update k1: 'v1'
```

```
  resp
end
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK generates an ID for it and records the start time and end time.

**Example Subsegment with metadata**

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

# Add annotations and metadata to segments with the X-Ray SDK for Ruby

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

**Annotations** are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions (p. 59). Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

**Metadata** are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

In addition to annotations and metadata, you can also record user ID strings (p. 265) on segments. User IDs are recorded in a separate field on segments and are indexed for use with search.

**Sections**



## Recording annotations with the X-Ray SDK for Ruby

Use annotations to record information on segments or subsegments that you want indexed for search.

**Annotation Requirements**

- **Keys** – Up to 500 alphanumeric characters. No spaces or symbols except underscores.
- **Values** – Up to 1,000 Unicode characters.
- **Entries** – Up to 50 annotations per trace.

**To record annotations**

1. Get a reference to the current segment or subsegment from `xray_recorder`.

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_segment
```

or

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_subsegment
```

2. Call `update` with a hash value.

```
my_annotations = { id: 12345 }
document.annotations.update my_annotations
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `add_annotations` twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotations.`*`key`* keyword in a filter expression (p. 59).

# Recording metadata with the X-Ray SDK for Ruby

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be strings, numbers, Booleans, or any object that can be serialized into a JSON object or array.

**To record metadata**

1. Get a reference to the current segment or subsegment from `xray_recorder`.

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_segment
```

or

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_subsegment
```

2. Call `metadata` with a String key; a Boolean, Number, String, or Object value; and a String namespace.

```
my_metadata = {
  my_namespace: {
    key: 'value'
  }
}
subsegment.metadata my_metadata
```

Calling `metadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

# Recording user IDs with the X-Ray SDK for Ruby

Record user IDs on request segments to identify the user who sent the request.

**To record user IDs**

1. Get a reference to the current segment from `xray_recorder`.

```
require 'aws-xray-sdk'
...
document = XRay.recorder.current_segment
```

2. Set the user field on the segment to a String ID of the user who sent the request.

```
segment.user = 'U12345'
```

You can set the user in your controllers to record the user ID as soon as your application starts processing a request.

To find traces for a user ID, use the `user` keyword in a filter expression (p. 59).

# AWS X-Ray SDK for .NET

The X-Ray SDK for .NET is a library for instrumenting C# .NET web applications, .NET Core web applications, and .NET Core functions on AWS Lambda. It provides classes and methods for generating and sending trace data to the X-Ray daemon (p. 137). This includes information about incoming requests served by the application, and calls that the application makes to downstream AWS services, HTTP web APIs, and SQL databases.

> **Note**
> The X-Ray SDK for .NET is an open source project. You can follow the project and submit issues and pull requests on GitHub: github.com/aws/aws-xray-sdk-dotnet

For web applications, start by adding a message handler to your web configuration (p. 272) to trace incoming requests. The message handler creates a segment (p. 21) for each traced request, and completes the segment when the response is sent. While the segment is open you can use the SDK client's methods to add information to the segment and create subsegments to trace downstream calls. The SDK also automatically records exceptions that your application throws while the segment is open.

For Lambda functions called by an instrumented application or service, Lambda reads the tracing header (p. 26) and traces sampled requests automatically. For other functions, you can configure Lambda (p. 167) to sample and trace incoming requests. In either case, Lambda creates the segment and provides it to the X-Ray SDK.

> **Note**
> On Lambda, the X-Ray SDK is optional. If you don't use it in your function, your service map will still include a node for the Lambda service, and one for each Lambda function. By adding the SDK, you can instrument your function code to add subsegments to the function segment recorded by Lambda. See AWS Lambda and AWS X-Ray (p. 167) for more information.

Next, use the X-Ray SDK for .NET to instrument your AWS SDK for .NET clients (p. 274). Whenever you make a call to a downstream AWS service or resource with an instrumented client, the SDK records information about the call in a subsegment. AWS services and the resources that you access within the services appear as downstream nodes on the service map to help you identify errors and throttling issues on individual connections.

The X-Ray SDK for .NET also provides instrumentation for downstream calls to HTTP web APIs (p. 276) and SQL databases (p. 277). The `GetResponseTraced` extension method for `System.Net.HttpWebRequest` traces outgoing HTTP calls. You can use the X-Ray SDK for .NET's version of `SqlCommand` to instrument SQL queries.

Once you get going with the SDK, customize its behavior by configuring the recorder and message handler (p. 267). You can add plugins to record data about the compute resources running your application, customize sampling behavior by defining sampling rules, and set the log level to see more or less information from the SDK in your application logs.

Record additional information about requests and the work that your application does in annotations and metadata (p. 280). Annotations are simple key-value pairs that are indexed for use with filter expressions (p. 59), so that you can search for traces that contain specific data. Metadata entries are less restrictive and can record entire objects and arrays — anything that can be serialized into JSON.

> **Annotations and Metadata**
> Annotations and metadata are arbitrary text that you add to segments with the X-Ray SDK. Annotations are indexed for use with filter expressions. Metadata are not indexed, but can be

viewed in the raw segment with the X-Ray console or API. Anyone that you grant read access to X-Ray can view this data.

When you have many instrumented clients in your code, a single request segment can contain a large number of subsegments, one for each call made with an instrumented client. You can organize and group subsegments by wrapping client calls in custom subsegments (p. 279). You can create a custom subsegment for an entire function or any section of code, and record metadata and annotations on the subsegment instead of writing everything on the parent segment.

For reference documentation about the SDK's classes and methods, see the following:

- AWS X-Ray SDK for .NET API Reference
- AWS X-Ray SDK for .NET Core API Reference

The same package supports both .NET and .NET Core, but the classes that are used vary. Examples in this chapter link to the .NET API reference unless the class is specific to .NET Core.

# Requirements

The X-Ray SDK for .NET requires the .NET framework and AWS SDK for .NET.

For .NET Core applications and functions, the SDK requires .NET Core 2.0 or later.

# Adding the X-Ray SDK for .NET to your application

Use NuGet to add the X-Ray SDK for .NET to your application.

**To install the X-Ray SDK for .NET with NuGet package manager in Visual Studio**

1. Choose **Tools**, **NuGet Package Manager**, **Manage NuGet Packages for Solution**.
2. Search for **AWSXRayRecorder**.
3. Choose the package, and then choose **Install**.

# Configuring the X-Ray SDK for .NET

You can configure the X-Ray SDK for .NET with plugins to include information about the service that your application runs on, modify the default sampling behavior, or add sampling rules that apply to requests to specific paths.

For .NET web applications, add keys to the `appSettings` section of your `Web.config` file.

**Example Web.config**

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin"/>
    <add key="SamplingRuleManifest" value="sampling-rules.json"/>
  </appSettings>
</configuration>
```

For .NET Core, create a file named `appsettings.json` with a top-level key named `XRay`.

**Example .NET appsettings.json**

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin",
    "SamplingRuleManifest": "sampling-rules.json"
  }
}
```

Then, in your application code, build a configuration object and use it to initialize the X-Ray recorder. Do this before you initialize the recorder (p. 273).

**Example .NET Core Program.cs – Recorder configuration**

```
using Amazon.XRay.Recorder.Core;
...
AWSXRayRecorder.InitializeInstance(configuration);
```

If you are instrumenting a .NET Core web application, you can also pass the configuration object to the `UseXRay` method when you configure the message handler (p. 273). For Lambda functions, use the `InitializeInstance` method as shown above.

For more information on the .NET Core configuration API, see Configure an ASP.NET Core App on docs.microsoft.com.

**Sections**



# Plugins

Use plugins to add data about the service that is hosting your application.

**Plugins**

- Amazon EC2 – `EC2Plugin` adds the instance ID, Availability Zone, and the CloudWatch Logs Group.
- Elastic Beanstalk – `ElasticBeanstalkPlugin` adds the environment name, version label, and deployment ID.
- Amazon ECS – `ECSPlugin` adds the container ID.

To use a plugin, configure the X-Ray SDK for .NET client by adding the `AWSXRayPlugins` setting. If multiple plugins apply to your application, specify all of them in the same setting, separated by commas.

**Example Web.config - plugins**

```
<configuration>
  <appSettings>
```

```
    <add key="AWSXRayPlugins" value="EC2Plugin,ElasticBeanstalkPlugin"/>
  </appSettings>
</configuration>
```

**Example .NET Core appsettings.json – Plugins**

```
{
  "XRay": {
    "AWSXRayPlugins": "EC2Plugin,ElasticBeanstalkPlugin"
  }
}
```

# Sampling rules

The SDK uses the sampling rules you define in the X-Ray console to determine which requests to record. The default rule traces the first request each second, and five percent of any additional requests across all services sending traces to X-Ray. Create additional rules in the X-Ray console (p. 70) to customize the amount of data recorded for each of your applications.

The SDK applies custom rules in the order in which they are defined. If a request matches multiple custom rules, the SDK applies only the first rule.

> **Note**
> If the SDK can't reach X-Ray to get sampling rules, it reverts to a default local rule of the first request each second, and five percent of any additional requests per host. This can occur if the host doesn't have permission to call sampling APIs, or can't connect to the X-Ray daemon, which acts as a TCP proxy for API calls made by the SDK.

You can also configure the SDK to load sampling rules from a JSON document. The SDK can use local rules as a backup for cases where X-Ray sampling is unavailable, or use local rules exclusively.

**Example sampling-rules.json**

```
{
  "version": 2,
  "rules": [
    {
      "description": "Player moves.",
      "host": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    }
  ],
  "default": {
    "fixed_target": 1,
    "rate": 0.1
  }
}
```

This example defines one custom rule and a default rule. The custom rule applies a five-percent sampling rate with no minimum number of requests to trace for paths under `/api/move/`. The default rule traces the first request each second and 10 percent of additional requests.

The disadvantage of defining rules locally is that the fixed target is applied by each instance of the recorder independently, instead of being managed by the X-Ray service. As you deploy more hosts, the fixed rate is multiplied, making it harder to control the amount of data recorded.

On AWS Lambda, you cannot modify the sampling rate. If your function is called by an instrumented service, calls that generated requests that were sampled by that service will be recorded by Lambda. If active tracing is enabled and no tracing header is present, Lambda makes the sampling decision.

To configure backup rules, tell the X-Ray SDK for .NET to load sampling rules from a file with the `SamplingRuleManifest` setting.

**Example .NET Web.config - sampling rules**

```
<configuration>
  <appSettings>
    <add key="SamplingRuleManifest" value="sampling-rules.json"/>
  </appSettings>
</configuration>
```

**Example .NET Core appsettings.json – Sampling rules**

```
{
  "XRay": {
    "SamplingRuleManifest": "sampling-rules.json"
  }
}
```

To use only local rules, build the recorder with a `LocalizedSamplingStrategy`. If you have backup rules configured, remove that configuration.

**Example .NET global.asax – Local sampling rules**

```
var recorder = new AWSXRayRecorderBuilder().WithSamplingStrategy(new
 LocalizedSamplingStrategy(samplingrules.json)).Build();
AWSXRayRecorder.InitializeInstance(recorder);
```

**Example .NET Core Program.cs – Local sampling rules**

```
var recorder = new AWSXRayRecorderBuilder().WithSamplingStrategy(new
 LocalizedSamplingStrategy(sampling-rules.json)).Build();
AWSXRayRecorder.InitializeInstance(configuration,recorder);
```

# Logging (.NET)

The X-Ray SDK for .NET uses the same logging mechanism as the AWS SDK for .NET. If you already configured your application to log AWS SDK for .NET output, the same configuration applies to output from the X-Ray SDK for .NET.

To configure logging, add a configuration section named `aws` to your `App.config` file or `Web.config` file.

**Example Web.config - logging**

```
...
<configuration>
  <configSections>
    <section name="aws" type="Amazon.AWSSection, AWSSDK.Core"/>
  </configSections>
  <aws>
```

```
    <logging logTo="Log4Net"/>
  </aws>
</configuration>
```

For more information, see Configuring Your AWS SDK for .NET Application in the *AWS SDK for .NET Developer Guide*.

# Logging (.NET Core)

For .NET Core applications, the X-Ray SDK supports the logging options in the AWS SDK for .NET LoggingOptions enum. To configure logging, pass one of these options to the `RegisterLogger` method.

```
AWSXRayRecorder.RegisterLogger(LoggingOptions.Console);
```

For example, to use log4net, create a configuration file that defines the logger, the output format, and the file location.

**Example .NET Core log4net.config**

```
<?xml version="1.0" encoding="utf-8" ?>
<log4net>
  <appender name="FileAppender" type="log4net.Appender.FileAppender,log4net">
    <file value="c:\logs\sdk-log.txt" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%date [%thread] %level %logger - %message%newline" />
    </layout>
  </appender>
  <logger name="Amazon">
    <level value="DEBUG" />
    <appender-ref ref="FileAppender" />
  </logger>
</log4net>
```

Then, create the logger and apply the configuration in your program code.

**Example .NET Core Program.cs – Logging**

```
using log4net;
using Amazon.XRay.Recorder.Core;

class Program
{
  private static ILog log;
  static Program()
  {
    var logRepository = LogManager.GetRepository(Assembly.GetEntryAssembly());
    XmlConfigurator.Configure(logRepository, new FileInfo("log4net.config"));
    log = LogManager.GetLogger(typeof(Program));
    AWSXRayRecorder.RegisterLogger(LoggingOptions.Log4Net);
  }
  static void Main(string[] args)
  {
  ...
  }
}
```

For more information on configuring log4net, see Configuration on logging.apache.org.

## Environment variables

You can use environment variables to configure the X-Ray SDK for .NET. The SDK supports the following variables.

- `AWS_XRAY_TRACING_NAME` – Set a service name that the SDK uses for segments. Overrides the service name that you set on the servlet filter's segment naming strategy (p. 274).
- `AWS_XRAY_DAEMON_ADDRESS` – Set the host and port of the X-Ray daemon listener. By default, the SDK uses `127.0.0.1:2000` for both trace data (UDP) and sampling (TCP). Use this variable if you have configured the daemon to listen on a different port (p. 140) or if it is running on a different host.

  **Format**

  - **Same port** – *`address`*`:`*`port`*
  - **Different ports** – `tcp:`*`address`*`:`*`port`* `udp:`*`address`*`:`*`port`*
- `AWS_XRAY_CONTEXT_MISSING` – Set to `LOG_ERROR` to avoid throwing exceptions when your instrumented code attempts to record data when no segment is open.

  **Valid Values**

  - `RUNTIME_ERROR` – Throw a runtime exception (default).
  - `LOG_ERROR` – Log an error and continue.

  Errors related to missing segments or subsegments can occur when you attempt to use an instrumented client in startup code that runs when no request is open, or in code that spawns a new thread.

# Instrumenting incoming HTTP requests with the X-Ray SDK for .NET

You can use the X-Ray SDK to trace incoming HTTP requests that your application serves on an EC2 instance in Amazon EC2, AWS Elastic Beanstalk, or Amazon ECS.

Use a message handler to instrument incoming HTTP requests. When you add the X-Ray message handler to your application, the X-Ray SDK for .NET creates a segment for each sampled request. This segment includes timing, method, and disposition of the HTTP request. Additional instrumentation creates subsegments on this segment.

> **Note**
> For AWS Lambda functions, Lambda creates a segment for each sampled request. See AWS Lambda and AWS X-Ray (p. 167) for more information.

Each segment has a name that identifies your application in the service map. The segment can be named statically, or you can configure the SDK to name it dynamically based on the host header in the incoming request. Dynamic naming lets you group traces based on the domain name in the request, and apply a default name if the name doesn't match an expected pattern (for example, if the host header is forged).

> **Forwarded Requests**
> If a load balancer or other intermediary forwards a request to your application, X-Ray takes the client IP from the `X-Forwarded-For` header in the request instead of from the source IP in the IP packet. The client IP that is recorded for a forwarded request can be forged, so it should not be trusted.

The message handler creates a segment for each incoming request with an `http` block that contains the following information:

- **HTTP method** – GET, POST, PUT, DELETE, etc.
- **Client address** – The IP address of the client that sent the request.
- **Response code** – The HTTP response code for the completed request.
- **Timing** – The start time (when the request was received) and end time (when the response was sent).
- **User agent** — The `user-agent` from the request.
- **Content length** — The `content-length` from the response.

**Sections**



# Instrumenting incoming requests (.NET)

To instrument requests served by your application, call `RegisterXRay` in the `Init` method of your `global.asax` file.

**Example global.asax - message handler**

```
using System.Web.Http;
using Amazon.XRay.Recorder.Handlers.AspNet;

namespace SampleEBWebApplication
{
  public class MvcApplication : System.Web.HttpApplication
  {
    public override void Init()
    {
      base.Init();
      AWSXRayASPNET.RegisterXRay(this, "MyApp");
    }
  }
}
```

# Instrumenting incoming requests (.NET Core)

To instrument requests served by your application, call the `UseExceptionHandler`, `UseXRay`, and `UseStaticFiles` methods in the `Configure` method of your `Startup` class.

**Example Startup.cs**

```
using Microsoft.AspNetCore.Builder;

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
  {
    app.UseExceptionHandler("/Error");
    app.UseXRay("MyApp");
    app.UseStaticFiles();
    app.UseMVC();
  }
```

Always call `UseXRay` after `UseExceptionHandler` to record exceptions. If you use other middleware, enable it after you call `UseXRay`.

The `UseXRay` method can also take a configuration object (p. 267) as a second argument.

```
app.UseXRay("MyApp", configuration);
```

## Configuring a segment naming strategy

AWS X-Ray uses a *service name* to identify your application and distinguish it from the other applications, databases, external APIs, and AWS resources that your application uses. When the X-Ray SDK generates segments for incoming requests, it records your application's service name in the segment's name field (p. 103).

The X-Ray SDK can name segments after the hostname in the HTTP request header. However, this header can be forged, which could result in unexpected nodes in your service map. To prevent the SDK from naming segments incorrectly due to requests with forged host headers, you must specify a default name for incoming requests.

If your application serves requests for multiple domains, you can configure the SDK to use a dynamic naming strategy to reflect this in segment names. A dynamic naming strategy allows the SDK to use the hostname for requests that match an expected pattern, and apply the default name to requests that don't.

For example, you might have a single application serving requests to three subdomains– `www.example.com`, `api.example.com`, and `static.example.com`. You can use a dynamic naming strategy with the pattern `*.example.com` to identify segments for each subdomain with a different name, resulting in three service nodes on the service map. If your application receives requests with a hostname that doesn't match the pattern, you will see a fourth node on the service map with a fallback name that you specify.

To use the same name for all request segments, specify the name of your application when you initialize the message handler, as shown in the previous section (p. 273). This has the same effect as creating a `FixedSegmentNamingStrategy` and passing it to the `RegisterXRay` method.

```
AWSXRayASPNET.RegisterXRay(this, new FixedSegmentNamingStrategy("MyApp"));
```

**Note**
You can override the default service name that you define in code with the `AWS_XRAY_TRACING_NAME` environment variable (p. 272).

A dynamic naming strategy defines a pattern that hostnames should match, and a default name to use if the hostname in the HTTP request does not match the pattern. To name segments dynamically, create a `DynamicSegmentNamingStrategy` and pass it to the `RegisterXRay` method.

```
AWSXRayASPNET.RegisterXRay(this, new DynamicSegmentNamingStrategy("MyApp",
 "*.example.com"));
```

# Tracing AWS SDK calls with the X-Ray SDK for .NET

When your application makes calls to AWS services to store data, write to a queue, or send notifications, the X-Ray SDK for .NET tracks the calls downstream in subsegments (p. 279). Traced AWS services and resources that you access within those services (for example, an Amazon S3 bucket or Amazon SQS queue), appear as downstream nodes on the service map in the X-Ray console.

You can instrument all of your AWS SDK for .NET clients by calling `RegisterXRayForAllServices` before you create them.

### Example SampleController.cs - DynamoDB client instrumentation

```
using Amazon;
using Amazon.Util;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.AwsSdk;

namespace SampleEBWebApplication.Controllers
{
  public class SampleController : ApiController
  {
    AWSSDKHandler.RegisterXRayForAllServices();
    private static readonly Lazy<AmazonDynamoDBClient> LazyDdbClient = new
 Lazy<AmazonDynamoDBClient>(() =>
    {
      var client = new AmazonDynamoDBClient(EC2InstanceMetadata.Region ??
 RegionEndpoint.USEast1);
      return client;
    });
```

To instrument clients for some services and not others, call `RegisterXRay` instead of `RegisterXRayForAllServices`. Replace the highlighted text with the name of the service's client interface.

```
AWSSDKHandler.RegisterXRay<IAmazonDynamoDB>()
```

For all services, you can see the name of the API called in the X-Ray console. For a subset of services, the X-Ray SDK adds information to the segment to provide more granularity in the service map.

For example, when you make a call with an instrumented DynamoDB client, the SDK adds the table name to the segment for calls that target a table. In the console, each table appears as a separate node in the service map, with a generic DynamoDB node for calls that don't target a table.

### Example Subsegment for a call to DynamoDB to save an item

```
{
  "id": "24756640c0d0978a",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "DynamoDB",
  "namespace": "aws",
  "http": {
    "response": {
      "content_length": 60,
      "status": 200
    }
  },
  "aws": {
    "table_name": "scorekeep-user",
    "operation": "UpdateItem",
    "request_id": "UBQNSO5AEM8T4FDA4RQDEB94OVTDRVV4K4HIRGVJF66Q9ASUAAJG",
  }
}
```

When you access named resources, calls to the following services create additional nodes in the service map. Calls that don't target specific resources create a generic node for the service.

- **Amazon DynamoDB** – Table name

- **Amazon Simple Storage Service** – Bucket and key name
- **Amazon Simple Queue Service** – Queue name

# Tracing calls to downstream HTTP web services with the X-Ray SDK for .NET

When your application makes calls to microservices or public HTTP APIs, you can use the X-Ray SDK for .NET's `GetResponseTraced` extension method for `System.Net.HttpWebRequest` to instrument those calls and add the API to the service graph as a downstream service.

**Example HttpWebRequest**

```
using System.Net;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.System.Net;

private void MakeHttpRequest()
{
  HttpWebRequest request = (HttpWebRequest)WebRequest.Create("http://names.example.com/
api");
  request.GetResponseTraced();
}
```

For asynchronous calls, use `GetAsyncResponseTraced`.

```
request.GetAsyncResponseTraced();
```

If you use `system.net.http.httpclient`, use the `HttpClientXRayTracingHandler` delegating handler to record calls.

**Example HttpClient**

```
using System.Net.Http;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.System.Net;

private void MakeHttpRequest()
{
  var httpClient = new HttpClient(new HttpClientXRayTracingHandler(new
 HttpClientHandler()));
  httpClient.GetAsync(URL);
}
```

When you instrument a call to a downstream web API, the X-Ray SDK for .NET records a subsegment with information about the HTTP request and response. X-Ray uses the subsegment to generate an inferred segment for the API.

**Example Subsegment for a downstream HTTP call**

```
{
  "id": "004f72be19cddc2a",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "name": "names.example.com",
  "namespace": "remote",
```

```
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  }
}
```

**Example Inferred segment for a downstream HTTP call**

```
{
  "id": "168416dc2ea97781",
  "name": "names.example.com",
  "trace_id": "1-5880168b-fd5153bb58284b67678aa78c",
  "start_time": 1484786387.131,
  "end_time": 1484786387.501,
  "parent_id": "004f72be19cddc2a",
  "http": {
    "request": {
      "method": "GET",
      "url": "https://names.example.com/"
    },
    "response": {
      "content_length": -1,
      "status": 200
    }
  },
  "inferred": true
}
```

# Tracing SQL queries with the X-Ray SDK for .NET

The X-Ray SDK for .NET provides a wrapper class for `System.Data.SqlClient.SqlCommand`, named `TraceableSqlCommand`, that you can use in place of `SqlCommand`. You can initialize an SQL command with the `TraceableSqlCommand` class.

## Tracing SQL queries with synchronous and asynchronous methods

The following examples show how to use the `TraceableSqlCommand` to automatically trace SQL Server queries synchronously and asynchronously.

**Example `Controller.cs` - SQL client instrumentation (synchronous)**

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;

private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
```

```
  using (var sqlCommand = new TraceableSqlCommand("SELECT " + id, sqlConnection))
  {
    sqlCommand.Connection.Open();
    sqlCommand.ExecuteNonQuery();
  }
}
```

You can execute the query asynchronously by using the `ExecuteReaderAsync` method.

**Example `Controller.cs` - SQL client instrumentation (asynchronous)**

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;
private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var sqlCommand = new TraceableSqlCommand("SELECT " + id, sqlConnection))
  {
    await sqlCommand.ExecuteReaderAsync();
  }
}
```

# Collecting SQL queries made to SQL Server

You can enable the capture of `SqlCommand.CommandText` as part of the subsegment created by your SQL query. `SqlCommand.CommandText` appears as the field `sanitized_query` in the subsegment JSON. By default, this feature is disabled for security.

> **Note**
> Do not enable the collection feature if you are including sensitive information as clear text in your SQL queries.

You can enable the collection of SQL queries in two ways:

- Set the `CollectSqlQueries` property to `true` in the global configuration for your application.
- Set the `collectSqlQueries` parameter in the `TraceableSqlCommand` instance to `true` to collect calls within the instance.

## Enable the global CollectSqlQueries property

The following examples show how to enable the `CollectSqlQueries` property for .NET and .NET Core.

.NET

To set the `CollectSqlQueries` property to `true` in the global configuration of your application in .NET, modify the `appsettings` of your `App.config` or `Web.config` file, as shown.

**Example `App.config` Or `Web.config` – Enable SQL Query collection globally**

```
<configuration>
<appSettings>
    <add key="CollectSqlQueries" value="true">
</appSettings>
</configuration>
```

.NET Core

To set the `CollectSqlQueries` property to `true` in the global configuration of your application in .NET Core, modify your `appsettings.json` file under the X-Ray key, as shown.

**Example `appsettings.json` – Enable SQL Query collection globally**

```
{
  "XRay": {
    "CollectSqlQueries":"true"
  }
}
```

## Enable the collectSqlQueries parameter

You can set the `collectSqlQueries` parameter in the `TraceableSqlCommand` instance to `true` to collect the SQL query text for SQL Server queries made using that instance. Setting the parameter to `false` disables the `CollectSqlQuery` feature for the `TraceableSqlCommand` instance.

**Note**
The value of `collectSqlQueries` in the `TraceableSqlCommand` instance overrides the value set in the global configuration of the `CollectSqlQueries` property.

**Example Example `Controller.cs` – Enable SQL Query collection for the instance**

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handlers.SqlServer;

private void QuerySql(int id)
{
  var connectionString = ConfigurationManager.AppSettings["RDS_CONNECTION_STRING"];
  using (var sqlConnection = new SqlConnection(connectionString))
  using (var command = new TraceableSqlCommand("SELECT " + id, sqlConnection,
 collectSqlQueries: true))
  {
    command.ExecuteNonQuery();
  }
}
```

# Creating additional subsegments

Subsegments extend a trace's segment (p. 21) with details about work done in order to serve a request. Each time you make a call with an instrumented client, the X-Ray SDK records the information generated in a subsegment. You can create additional subsegments to group other subsegments, to measure the performance of a section of code, or to record annotations and metadata.

To manage subsegments, use the `BeginSubsegment` and `EndSubsegment` methods. Perform any work in the subsegment in a `try` block and use `AddException` to trace exceptions. Call `EndSubsegment` in a `finally` block to ensure that the subsegment is closed.

**Example Controller.cs – Custom subsegment**

```
AWSXRayRecorder.Instance.BeginSubsegment("custom method");
try
```

```
{
  DoWork();
}
catch (Exception e)
{
  AWSXRayRecorder.Instance.AddException(e);
}
finally
{
  AWSXRayRecorder.Instance.EndSubsegment();
}
```

When you create a subsegment within a segment or another subsegment, the X-Ray SDK for .NET generates an ID for it and records the start time and end time.

**Example Subsegment with metadata**

```
"subsegments": [{
  "id": "6f1605cd8a07cb70",
  "start_time": 1.480305974194E9,
  "end_time": 1.4803059742E9,
  "name": "Custom subsegment for UserModel.saveUser function",
  "metadata": {
    "debug": {
      "test": "Metadata string from UserModel.saveUser"
    }
  },
```

# Add annotations and metadata to segments with the X-Ray SDK for .NET

You can record additional information about requests, the environment, or your application with annotations and metadata. You can add annotations and metadata to the segments that the X-Ray SDK creates, or to custom subsegments that you create.

**Annotations** are key-value pairs with string, number, or Boolean values. Annotations are indexed for use with filter expressions (p. 59). Use annotations to record data that you want to use to group traces in the console, or when calling the `GetTraceSummaries` API.

**Metadata** are key-value pairs that can have values of any type, including objects and lists, but are not indexed for use with filter expressions. Use metadata to record additional data that you want stored in the trace but don't need to use with search.

**Sections**



## Recording annotations with the X-Ray SDK for .NET

Use annotations to record information on segments or subsegments that you want indexed for search.

**Annotation Requirements**

- **Keys** – Up to 500 alphanumeric characters. No spaces or symbols except underscores.

- **Values** – Up to 1,000 Unicode characters.
- **Entries** – Up to 50 annotations per trace.

### To record annotations

1. Get an instance of `AWSXRayRecorder`.

```
using Amazon.XRay.Recorder.Core;
...
AWSXRayRecorder recorder = AWSXRayRecorder.Instance;
```

2. Call `addAnnotation` with a String key and a Boolean, Int32, Int64, Double, or String value.

```
recorder.AddAnnotation("mykey", "my value");
```

The SDK records annotations as key-value pairs in an `annotations` object in the segment document. Calling `addAnnotation` twice with the same key overwrites previously recorded values on the same segment or subsegment.

To find traces that have annotations with specific values, use the `annotations.`*`key`* keyword in a filter expression (p. 59).

# Recording metadata with the X-Ray SDK for .NET

Use metadata to record information on segments or subsegments that you don't need indexed for search. Metadata values can be Strings, Numbers, Booleans, or any other Object that can be serialized into a JSON object or array.

### To record metadata

1. Get an instance of `AWSXRayRecorder`.

```
using Amazon.XRay.Recorder.Core;
...
AWSXRayRecorder recorder = AWSXRayRecorder.Instance;
```

2. Call `AddMetadata` with a String namespace, String key, and an Object value.

```
segment.AddMetadata("my namespace", "my key", "my value");
```

   or

   Call `putMetadata` with just a key and value.

```
segment.AddMetadata("my key", "my value");
```

If you don't specify a namespace, the SDK uses `default`. Calling `AddMetadata` twice with the same key overwrites previously recorded values on the same segment or subsegment.

# Troubleshooting AWS X-Ray

This topic lists common errors and issues that you might encounter when using the X-Ray API, console, or SDKs. If you find an issue that is not listed here, you can use the **Feedback** button on this page to report it.

**Sections**



## X-Ray SDK for Java

**Error:** *Exception in thread "Thread-1" com.amazonaws.xray.exceptions.SegmentNotFoundException: Failed to begin subsegment named 'AmazonSNS': segment cannot be found.*

This error indicates that the X-Ray SDK attempted to record an outgoing call to AWS, but couldn't find an open segment. This can occur in the following situations:

- **A servlet filter is not configured** – The X-Ray SDK creates segments for incoming requests with a filter named `AWSXRayServletFilter`. Configure a servlet filter (p. 196) to instrument incoming requests.
- **You're using instrumented clients outside of servlet code** – If you use an instrumented client to make calls in startup code or other code that doesn't run in response to an incoming request, you must create a segment manually. See Instrumenting startup code (p. 128) for examples.
- **You're using instrumented clients in worker threads** – When you create a new thread, the X-Ray recorder loses its reference to the open segment. You can use the `getTraceEntity` and `setTraceEntity` methods to get a reference to the current segment or subsegment (`Entity`), and pass it back to the recorder inside of the thread. See Using instrumented clients in worker threads (p. 134) for an example.

## X-Ray SDK for Node.js

**Issue:** *CLS does not work with Sequelize*

Pass the X-Ray SDK for Node.js namespace to Sequelize with the `cls` method.

```
var AWSXRay = require('aws-xray-sdk');
const Sequelize = require('sequelize');
Sequelize.cls = AWSXRay.getNamespace();
const sequelize = new Sequelize('database', 'username', 'password');
```

**Issue:** *CLS does not work with Bluebird*

Use `cls-bluebird` to get Bluebird working with CLS.

```
var AWSXRay = require('aws-xray-sdk');
var Promise = require('bluebird');
var clsBluebird = require('cls-bluebird');
clsBluebird(AWSXRay.getNamespace());
```

# The X-Ray daemon

**Issue:** *The daemon is using the wrong credentials*

The daemon uses the AWS SDK to load credentials. If you use multiple methods of providing credentials, the method with the highest precedence is used. See Running the daemon (p. 138) for more information.