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 client 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 services and HTTP web APIs.

> **Note**
> This is prerelease documentation for a service in preview release. It is subject to change. Sign up for the preview here.

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

The X-Ray SDKs provide **interceptors** that you can 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, and an **HTTP client** that you can use to instrument calls to other internal and external HTTP web services. Some of the SDKs also support instrumenting calls to SQL databases, automatic AWS SDK client instrumentation, and other features.

Instead of sending trace data directly to X-Ray, the SDKs send JSON segment documents to a daemon process listening for UDP traffic. The **X-Ray daemon (p. 82)** buffers segments in a queue and uploads them to X-Ray in batches. The daemon is available for Linux and Windows and is included on AWS Elastic Beanstalk platforms.

Use the getting started tutorial (p. 5) to start using X-Ray in just a few minutes with an instrumented sample application, or keep reading (p. 2) to learn about the languages, frameworks and services that work with X-Ray.

Using AWS X-Ray

Use the X-Ray SDK and AWS service integration to instrument requests to your applications running on Amazon EC2, Elastic Beanstalk, or Amazon ECS.

To instrument your application code, you can 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 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, Elastic Beanstalk platforms include the X-Ray daemon and run it for you.

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 by adding the X-Ray SDK for Java's filter to your servlet configuration, and taking the AWS SDK 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.

Java

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

See The AWS X-Ray SDK for Java (p. 42) for more information.

- **Tomcat** – Add a servlet filter to your deployment descriptor (`web.xml`) to instrument incoming requests.
- **Spring Boot** – Add a servlet filter to your `WebConfig` class to instrument incoming requests.
- **Other frameworks** – Add a servlet filter if your framework supports servlets, or manually create segments and make sampling decisions on incoming requests if it doesn't.

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, and outgoing HTTP calls. Automatic request instrumentation is available for applications that use the Express framework.

See The X-Ray SDK for Node.js (p. 64) for more information.

- **Express** – Use the X-Ray SDK for Node.js Express middleware to instrument incoming requests.
- **Other frameworks** – Manually create segments and make sampling decisions on incoming requests.

C#

On Windows Server editions other than Windows Server Core, you can use the X-Ray SDK for .NET to instrument incoming requests, AWS SDK clients, and outgoing HTTP calls.

See The AWS X-Ray SDK for .NET (p. 73) for more information.

- **.NET on Windows Server** – Add a message handler to your HTTP configuration to instrument incoming requests.

If the X-Ray SDK is not 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. 21).

Supported AWS Services

Several AWS services provide **X-Ray integration**. Integrated services (p. 93) 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 AWS X-Ray daemon to receive segments from the X-Ray SDK.

Services with X-Ray integration include:

- **Amazon API Gateway** – Request tracing. API Gateway passes the trace ID to AWS Lambda and adds it to the request header for other downstream services.
- **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.
- **AWS Elastic Beanstalk** – Tooling. Elastic Beanstalk includes the X-Ray daemon on the following platforms:
  - **Java SE** – 2.3.0 and newer configurations
  - **Tomcat** – 2.4.0 and newer configurations

- **Node.js** – 3.2.0 and newer configurations
- **Windows Server** – All configurations other than Windows Server Core released since December 9th, 2016.

You can tell Elastic Beanstalk to run the daemon on the above platforms in the Elastic Beanstalk console, or by with the `XRayEnabled` option in the `aws:elasticbeanstalk:xray` namespace.

Code and Configuration Changes

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

Examples of Code and Configuration Changes

- **AWS resource configuration** – 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 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** – Import the X-Ray SDK for Java version of `HttpClientBuilder` instead of the Apache.org version to instrument outgoing HTTP calls in Java.
- **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, 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. 42) 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. 48) 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 one of the X-Ray managed policies 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.

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.

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

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-v1.1.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-v1.1.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 Graph in the X-Ray Console

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

To use the X-Ray console

1. Open 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.

Choose any node in the service graph to view traces for requests that generated traffic to that node. The X-Ray SDK for Java generates the trace data and sends it to the X-Ray daemon running on the same instance, which uploads trace data in batches to X-Ray.

For more insight into how each component works, you can use the Elastic Beanstalk console to view logs from the sample app and X-Ray daemon.

To view logs in the Elastic Beanstalk console

1. Open the Elastic Beanstalk console.
2. Navigate to the management console for your environment.
3. Choose **Logs**.
4. Choose **Request Logs**, and then choose **Last 100 Lines**.
5. When the logs appear, choose **Download**.

Locate the entry for `/var/log/xray-daemon.log`. This is the log for the X-Ray daemon that aggregates traces and uploads them to X-Ray. If there is an error posting traces, details appear here.

Example /var/log/xray-daemon.log

```
2016-12-03T00:09:02Z [Info] Initializing AWS X-Ray daemon 1.0.1
2016-12-03T00:09:02Z [Info] Using memory limit of 49 MB
2016-12-03T00:09:02Z [Info] 313 segment buffers allocated
2016-12-03T00:10:08Z [Info] Successfully sent batch of 1 segments (0.024
 seconds)
2016-12-03T00:10:09Z [Info] Successfully sent batch of 2 segments (0.042
 seconds)
2016-12-03T00:10:10Z [Info] Successfully sent batch of 7 segments (0.007
 seconds)
```

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:1.0.4-beta")
    }
}
```

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.strategy.sampling.DefaultSamplingStrategy;

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

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

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

This example uses the builder to load sampling rules from a file named `sampling-rules.json`

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

```
{
  "rules": {
    "user": {
      "id": 1,
      "service_name": "*",
      "http_method": "POST",
      "url_path": "/api/user",
      "fixed_target": 10,
      "rate": 1.0
    },
    "move": {
      "id": 2,
      "service_name": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 1,
      "rate": 0.05
    },
    "base": {
      "id": 3,
      "service_name": "*",
      "http_method": "*",
      "url_path": "*",
      "fixed_target": 10,
      "rate": 0.25
    }
  }
}
```

This example defines two path based rules and overrides the default rule. The first rule applies a 100% sampling rate to new user creations with POST requests to `/api/user`. The second rule traces the

first move request received each second, and then applies a 5% sampling rate to additional moves. The final rule overrides the default sampling rule with a rule that traces the first 10 requests each second and 25 percent of additional requests.

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 X-Ray SDK for Java Sample Application (p. 48).

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. 13).

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** – The AWS X-Ray SDK for Java (p. 42)
- **X-Ray SDK for Node.js** – The X-Ray SDK for Node.js (p. 64)
- **X-Ray SDK for .NET** – The AWS X-Ray SDK for .NET (p. 73)

To run the X-Ray daemon locally or on AWS, see The AWS X-Ray Daemon (p. 82).

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

AWS X-Ray Concepts

AWS X-Ray receives data from services in the form of *segments*, groups segments with a common request into *traces*, and processes traces to generate a *service graph* and provide a visual representation of your application.

Concepts

Segments

The compute resources running your application logic send data about the work that they do in the form of **segments**. A segment provides the name of the resource, details about the request, and details about the work done. For example, when an HTTP request reaches your application, it can record data about:

- **The host** – host name, alias or IP address
- **The request** – method, client address, path, user agent
- **The response** – status, content
- **The work done** – start and end times, subsegments

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 which data is collected by adding modifying your application configuration or code to instrument incoming requests, downstream requests, and AWS SDK clients.

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

Subsegments

Data about the work done can be broken down into **subsegments**, which 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.

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, with **edges** connecting the services that work together to serve requests.

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

For example, your application could use an application load balancer to distribute traffic to Amazon EC2 instances, which use the AWS SDK to contact DynamoDB to store data, and make HTTP calls to external web APIs. In the X-Ray service graph, the instances, DynamoDB tables, and downstream HTTP APIs all appear as separate services connected by edges.

Traces

The path of a request through your application is tracked with a **trace ID**. A trace collects all of the segments generated by a single request, 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. A trace ID header is added to each HTTP request by the first supported service that it interacts with, and propagates downstream to track the latency, disposition, and other request data.

Sampling

To ensure that tracing is efficient, while still providing a representative sample of the requests that your application serves, the first service that a request hits applies a **sampling** algorithm to determine which requests get traced. You can modify the default sampling rules and configure different sampling rates for different routes that your application serves with the X-Ray SDK.

Tracing Header

All requests are traced up to a configurable minimum, after which 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 tracing header is added to the request by the first X-Ray-integrated service that it hits, 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 ID of the segment 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.

The AWS X-Ray Console

The AWS X-Ray console lets you 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.

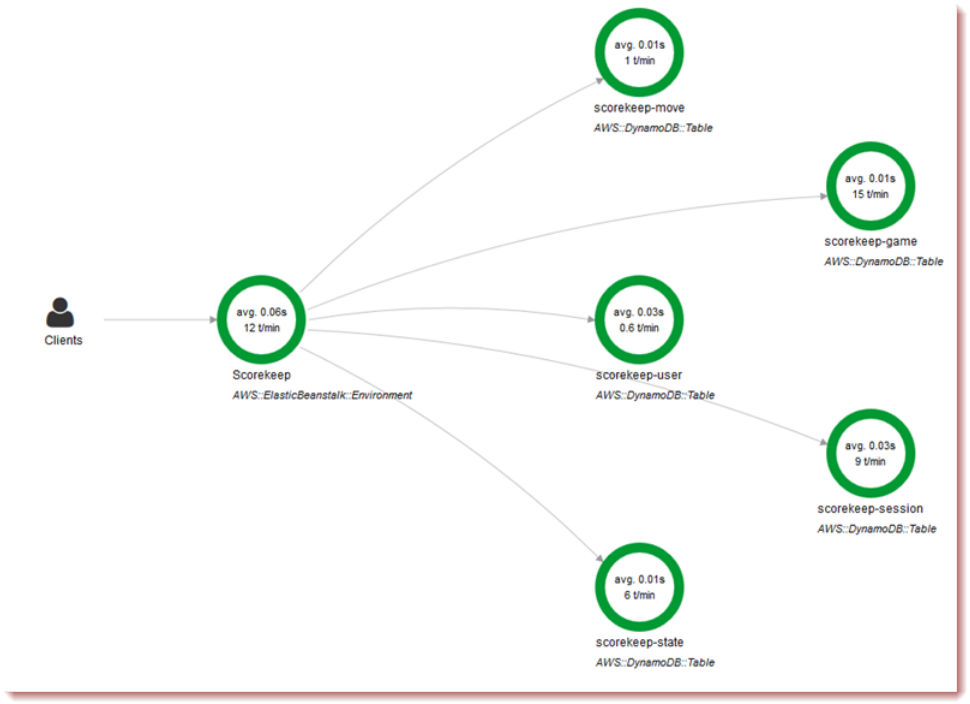
You can use filters to view a service map or traces for a specific request, a service, a connection between two services (an edge), or requests that satisfy a condition. X-Ray provides a filter expression language for filtering requests, services, and edgdes based on data in request headers, response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

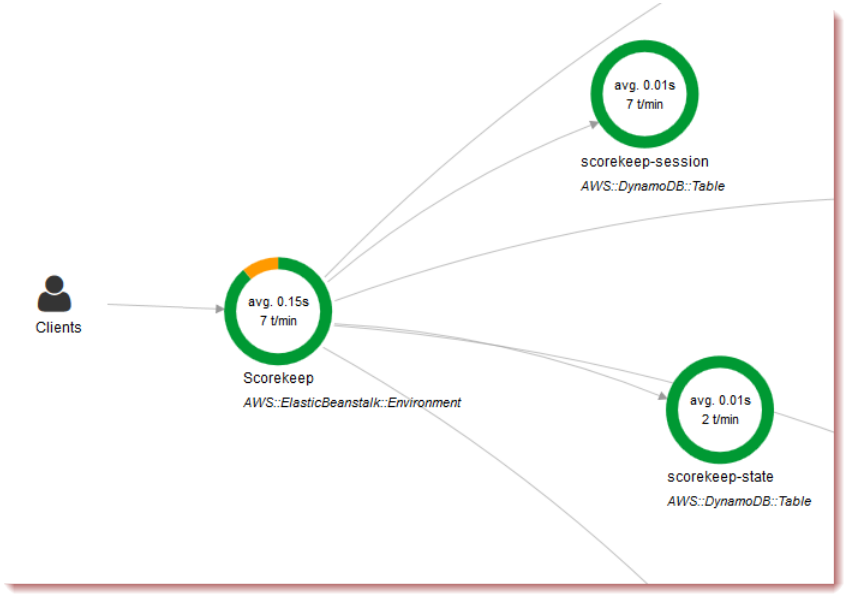


This service map shows a web API and five DynamoDB tables that it calls.

2. Choose a service node to view traces for that node, or an edge between two nodes to view traces for requests that travelled that connection.

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).



This service map shows a web API with 7% of requests returning 400 series errors.

Using Filter Expressions

Use filter expressions to view a service map or traces for requests that have performance issues or relate to specific requests.

Requests where response time was more than 5 seconds:

```
responsetime > 5
```

Request where the total duration was 5 to 8 seconds:

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

Requests that included a call to "api.example.com" with a fault (500 series error) or latency above 2.5 seconds, and one or more segments has an annotation named "account" with value "12345".

```
service("api.example.com") { fault = true OR responsetime > 2.5 } AND
 annotation.account = "12345"
```

Request where the service "api.example.com" made a call to "backend.example.com" that failed with a fault.

```
edge("api.example.com", "backend.example.com") { fault = true }
```

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")
```

Requests that completed successfully in under 3 seconds, including all downstream calls.

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

Boolean Keywords

- `ok` – Response status code was 2XX Success.
- `error` – Response status code was 4XX Client Error.
- `fault` – Response status code was 5XX Server Error.
- `partial` – Request has incomplete segments.

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.

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 user field on any segment in the trace.
- `annotation.`*`key`* – Value of annotation with field *`key`*.

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(`*`name`*`) {`*`filter`*`}` – Connection between services *`source`* and *`destination`*. Optional curly braces can contain a filter expression that applies to segments on this connection.

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

- Single trace - timeline: `xray/home#/traces/`*`trace-id`*
- Single trace - raw JSON: `xray/home#/traces/`*`trace-id`*`/raw`

Example: `xray/home#/traces/1-57f5498f-d91047849216d0f2ea3b6442/raw`

- Single trace - map: `xray/home#/traces/`*`trace-id`*`/map`

Filter Expressions

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

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

  Example (URL encoded): `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`

Time Range

Specify a length of time or start and end time in ISO8601 format.

- Length of time: `xray/home#time-range=`*`range-in-minutes`*

  Example - last minute: `xray/home#time-range=PT1M`
- Start and end time: `xray/home#time-range=`*`start`*`~`*`end`*

  Example: `xray/home#time-range=2016-11-06T01:05:00~2016-11-08T13:46:31`

  Example: `xray/home#time-range=2016-11-06T01:05~2016-11-08T13:46`

Combined

- Example: `xray/home#/traces?time-range=PT15M&filter=duration%20%3E%3D%205%20AND%20duration%20%3C%3D%208`
- Output:
  - View: Traces Section
  - Time Range: Last 15 Minutes
  - Filter: duration >= 5 AND duration <= 8

The 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.

Sections

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. 5) 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.

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-east-1.elasticbeanstalk.com
```

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,"state
["JQVLEOM2","D67QLPIC","VF9BM9NC","OEAA6GK9","2A705O73","1U2LFTLJ","HUKIDD70","BAN1C8FI"
["BS8F8LQ","4MTTSPKP","463OETES","SVEBCL3N","N7CQ1GHP","O84ONEPD","EG4BPROQ","V4BLIDJ3",
```

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.example.us-west-2.elasticbeanstalk.com"
            ],
            "ReferenceId": 1,
            "State": "active",
            "EndTime": 1479068651.0,
            "Root": true,
            "Name": "scorekeep.example.us-west-2.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.example.us-west-2.elasticbeanstalk.com/api/
game/6Q0UE1DG/5FGLM9U3/endtime/1479069438",
    "http://scorekeep.example.us-west-2.elasticbeanstalk.com/api/session/
KH4341QH",
    "http://scorekeep.example.us-west-2.elasticbeanstalk.com/api/game/
GLQBJ3K5/153AHDIA",
    "http://scorekeep.example.us-west-2.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.

Uploading Segment Documents

You can upload segments and subsegments with the `PutTraceSegments` API.

Required Segment Document Fields

- `name` – The name of the service that handled the request.
- `id` – A 64-bit identifier for the segment, unique among segments in the same trace, in 16 hexadecimal digits.

  Trace ID Security
  Trace IDs are visible in response headers (p. 14). 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.
- `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 2nd, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal.
  - A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.
- `start_time` – Time the segment or subsegment was created, in floating point seconds in epoch time, accurate to milliseconds. For example, `1480615200.010` or `1.480615200010E9`
- `end_time` – Time the segment or subsegment was closed. For example, `1480615200.090` or `1.480615200090E9`. Specify either an `end_time` or `in_progress`.
- `in_progress` – Set to `true` instead of specifying an `end_time` to record that a segment has been 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 fact that the request was received. When the response is sent, send the complete segment to overwrite the in-progress segment.

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

Example Minimal complete segment

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

Example In-progress segment

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

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 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. 27).

Sending Segment Documents to the X-Ray Daemon

You can send segments and subsegments to the X-Ray daemon, which will buffer them and upload to the X-Ray API in batches.

Send the segment in JSON over UDP port 2000, prepended by the daemon's 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":
 "hello-1.mbfzqxzcpe.us-east-1.elasticbeanstalk.com"}
```

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-beta.zip. 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 (p. 18).

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. 82). For more information, see Sending Segment Documents to the X-Ray Daemon (p. 27).

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 alphanumeric characters 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 2nd, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal.
  - A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.

    Trace ID Security
    Trace IDs are visible in response headers (p. 14). 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 generates the segment, but this is not enforced. Any application with permission to `PutTraceSegments` can send segments with any name.

The following fields are optional for subsegments.

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.
- `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. 14) for downstream HTTP calls.
- `http` – `http` (p. 33) objects with information about the original HTTP request.
- `aws` – `aws` (p. 37) object with information about the AWS resource on which your application served the request.

- `error`, `throttle`, `fault`, and `cause` – error (p. 39) fields that indicate an error occurred and that include information about the exception that caused the error.
- `annotations` – `annotations` (p. 35) object with key-value pairs that you want X-Ray to index for search.
- `metadata` – `metadata` (p. 36) object with any additional data that you want to store in the segment.
- `subsegments` – **array** of `subsegment` (p. 30) 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 2nd, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal.
- 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.
- `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. 33) object with information about an outgoing HTTP call.
- `aws` – `aws` (p. 37) object with information about the downstream AWS resource that your application called.
- `error`, `throttle`, `fault`, and `cause` – error (p. 39) fields that indicate an error occurred and that include information about the exception that caused the error.
- `annotations` – `annotations` (p. 35) object with key-value pairs that you want X-Ray to index for search.
- `metadata` – `metadata` (p. 36) object with any additional data that you want to store in the segment.
- `subsegments` – **array** of `subsegment` (p. 30) 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 requestor's client.
  - `client_ip` – The IP address of the requestor. 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"
    }
  },
  "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).

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",
  "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
    }
  }
```

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.

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": "1.0.4-beta for Java"
  }
}
```

- `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.
- `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 (400 series error).
- `throttle` – **boolean** indicating that a request was throttled (429 error).
- `fault` – **boolean** indicating that a server error occurred (500 series 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=?;"
  }
}
```

The 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.

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.

Start by adding `AWSXRayServletFilter` as a servlet filter (p. 54) to trace incoming requests. A servlet filter creates a segment to record information about each traced request that your application serves, and completes the segment when the response is sent. Within this segment you can create subsegments to trace downstream calls to other AWS services, HTTP web APIs, and SQL databases.

Next, use the X-Ray SDK for Java to instrument your SDK for Java clients (p. 77). 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.

When you include the SDK Instrumentor submodule (p. 43) in your build configuration, the X-Ray SDK for Java instruments all AWS SDK for Java clients automatically. Any call to an AWS service made with an instrumented client adds a subsegment to the request trace with information about the service, the resource in the service that was changed (if any), the response, and latency.

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. 57) 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's versions of `HTTPClient` and `HTTPClientBuilder` (p. 58) in the Apache HTTP submodule to instrument Apache HTTP clients. To instrument SQL queries, add the SDK's interceptor to your data source (p. 60).

The X-Ray SDK for Java is split into submodules for each type of trace data. The submodules and bill of materials are available from Maven:

- `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-instrumentor` – With `aws-xray-recorder-sdk-aws-sdk`, instruments all AWS SDK for Java clients automatically.
- `aws-xray-recorder-sdk-apache-http` – Instruments outbound HTTP calls made with Apache HTTP clients.
- `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.

For reference documentation for 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 1.11.42 or later
- Servlet API 3.0
- Jackson Core, Databind and Annotations 2.8
- Commons Validator 1.5.1

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 JARs in your deployment for that dependency, you must remove those JARs 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`
- **Bill of Materials** – `aws-xray-recorder-sdk-bom`
- **Version** – `1.0.4-beta`

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>1.0.2-beta</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
<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>
```

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:1.0.4-beta')
    }
}
```

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. 48) for an example that uses Gradle.

Configuring the X-Ray SDK for Java

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.

Sections

Service Plugins

Use `plugins` to add trace data about the service hosting your application.

- `Amazon EC2` – Adds the instance ID.
- `Elastic Beanstalk` – Adds the environment ID.
- `Amazon ECS` – Adds the cluster ID.

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.strategy.sampling.DefaultSamplingStrategy;

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

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

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

Sampling Rules

The SDK has a default sampling strategy that determines which requests get traced. By default, the SDK traces the first request each second, and 5 percent of any additional requests. You can customize the SDK's sampling behavior by applying rules defined in a local file.

Example sampling-rules.json

```
{
  "rules": {
    "move": {
      "id": 1,
      "service_name": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    "base": {
      "id": 2,
      "service_name": "*",
      "http_method": "*",
      "url_path": "*",
      "fixed_target": 1,
      "rate": 0.1
    }
  }
}
```

This example defines two rules. The first rule applies a five-percent sampling rate with no minimum number of requests to trace to requests with paths under `/api/move/`. The second overrides the default sampling rule with a rule that traces the first request each second and 10 percent of additional requests.

For Spring, configure the global recorder 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.DefaultSamplingStrategy;

@Configuration
public class WebConfig {

  static {
  AWSXRayRecorderBuilder builder =
 AWSXRayRecorderBuilder.standard().withPlugin(new EC2Plugin());

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

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

For Tomcat, add a listener that extends `ServletContextListener`.

Example src/com/myapp/web/Startup.java

```
package com.myapp.web;

import java.net.URL;

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import com.amazonaws.xray.AWSXRay;
import com.amazonaws.xray.AWSXRayRecorderBuilder;
import com.amazonaws.xray.plugins.EC2Plugin;
import com.amazonaws.xray.strategy.sampling.DefaultSamplingStrategy;

public class Startup implements ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent event) {
        AWSXRayRecorderBuilder builder =
 AWSXRayRecorderBuilder.standard().withPlugin(new EC2Plugin());

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

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

    @Override
    public void contextDestroyed(ServletContextEvent event) { }
}
```

Register the listener in the deployment descriptor.

Example WEB-INF/web.xml

```
...
  <listener>
    <listener-class>com.myapp.web.Startup</listener-class>
  </listener>
```

Logging

By default, the SDK outputs `SEVERE` 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

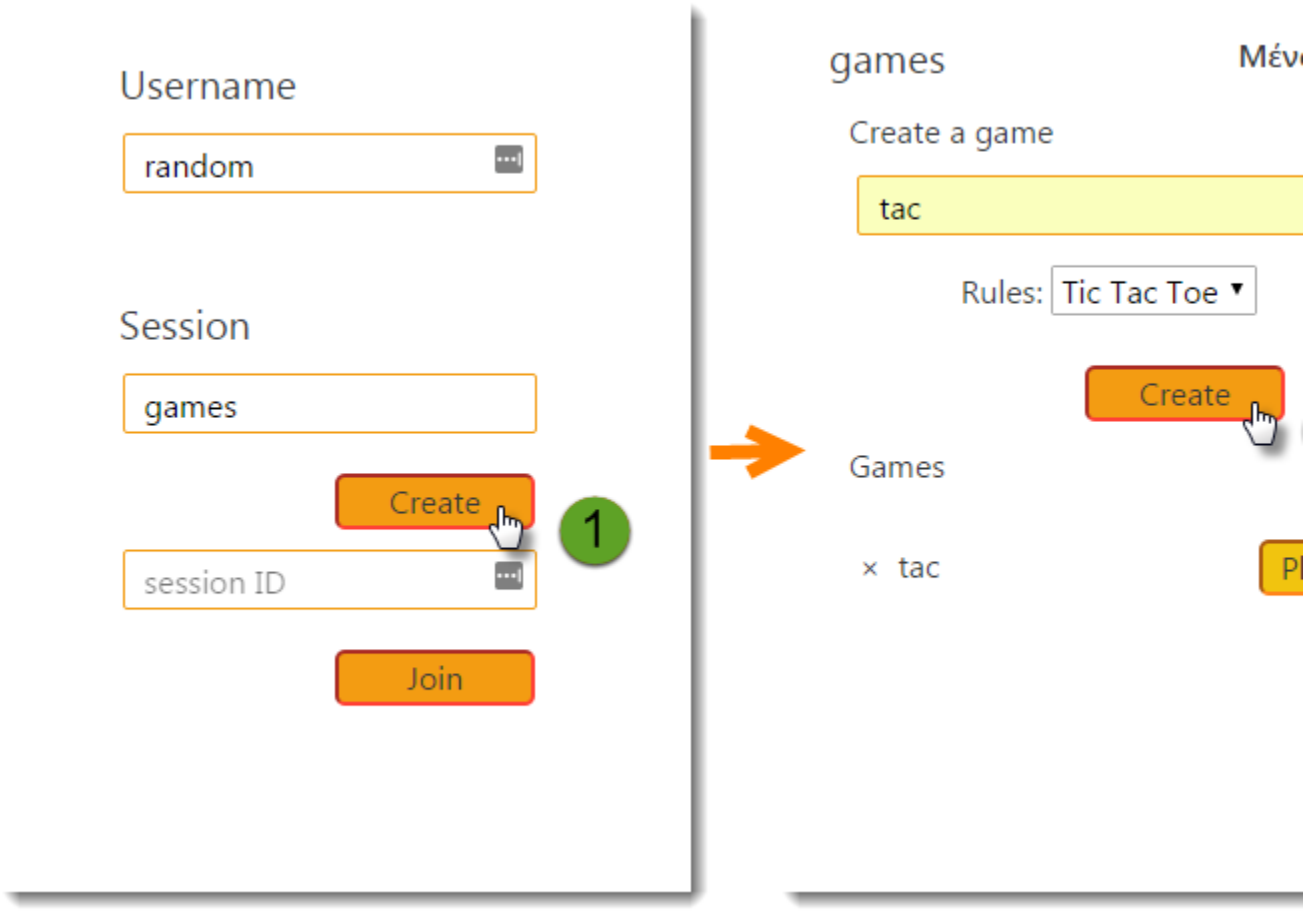
Set the logging level with the `logging.level.com.amazonaws.xray` property.

```
logging.level.com.amazonaws.xray = DEBUG
```

Use debug logs to identify issues such as unclosed subsegments when you generate subsegments manually (p. 62).

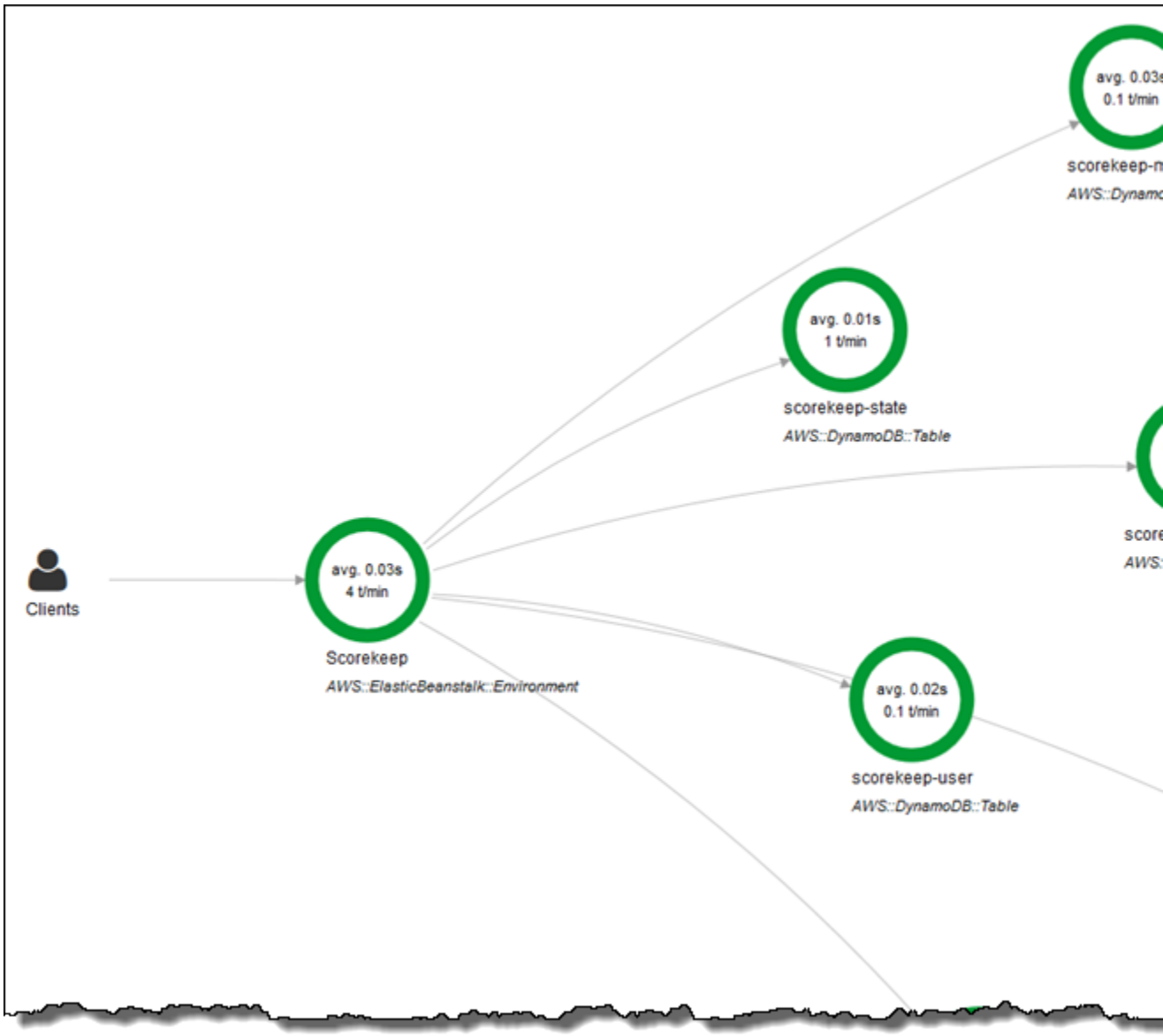
X-Ray SDK for Java Sample Application

A sample application that shows the use of the SDK to instrument incoming HTTP calls, DynamoDB SDK clients, and HTTP clients is available on GitHub. The app, eb-java-scorekeep, 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 of the components are hosted in an AWS Elastic Beanstalk environment for portability and ease of deployment.

The `xray` branch of the application shows the use of filters (p. 54), plugins (p. 45), instrumented AWS SDK clients (p. 57), HTTPClient (p. 58), SQL queries (p. 60), and custom subsegments (p. 62).



The sample application shows basic instrumentation in these files:

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

For instructions on using the sample application with X-Ray, see the getting started tutorial (p. 5). 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 (p. 51)
- Creating Additional Subsegments (p. 51)
- Instrumenting Outgoing HTTP Calls (p. 51)
- Instrumenting Calls to a PostgreSQL Database (p. 52)

Manually Instrumenting AWS SDK Clients

Sample – Manual AWS SDK Client Instrumentation – `SessionModel.java`

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. 43).

You can disable automatic client instrumentation by removing the Instrumentor submodule, which 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

```
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

Sample – Manual segment creation and annotation – `UserModel.java`

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 - saveUser

```
  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 {
      subsegment.putMetadata("debug", "test", "Metadata string from
 UserModel.saveUser");
      AWSXRay.endSubsegment();
    }
  }
```

Instrumenting Outgoing HTTP Calls

Sample – HTTPClient instrumentation – `UserFactory.java`

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

```
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`.

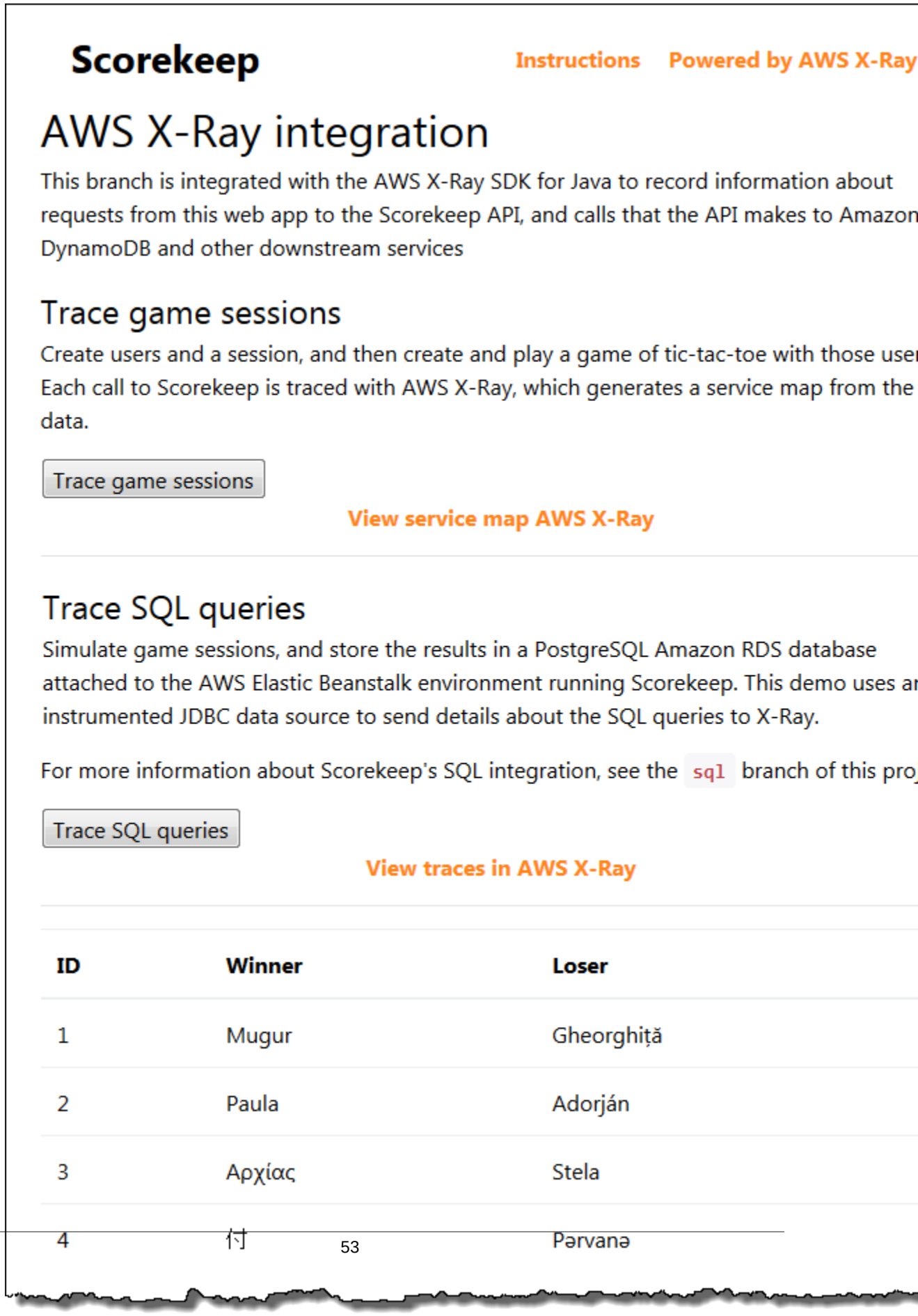
Instrumenting Calls to a PostgreSQL Database

Sample – PostgreSQL Database Instrumentation – `application-pgsql.properties`

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

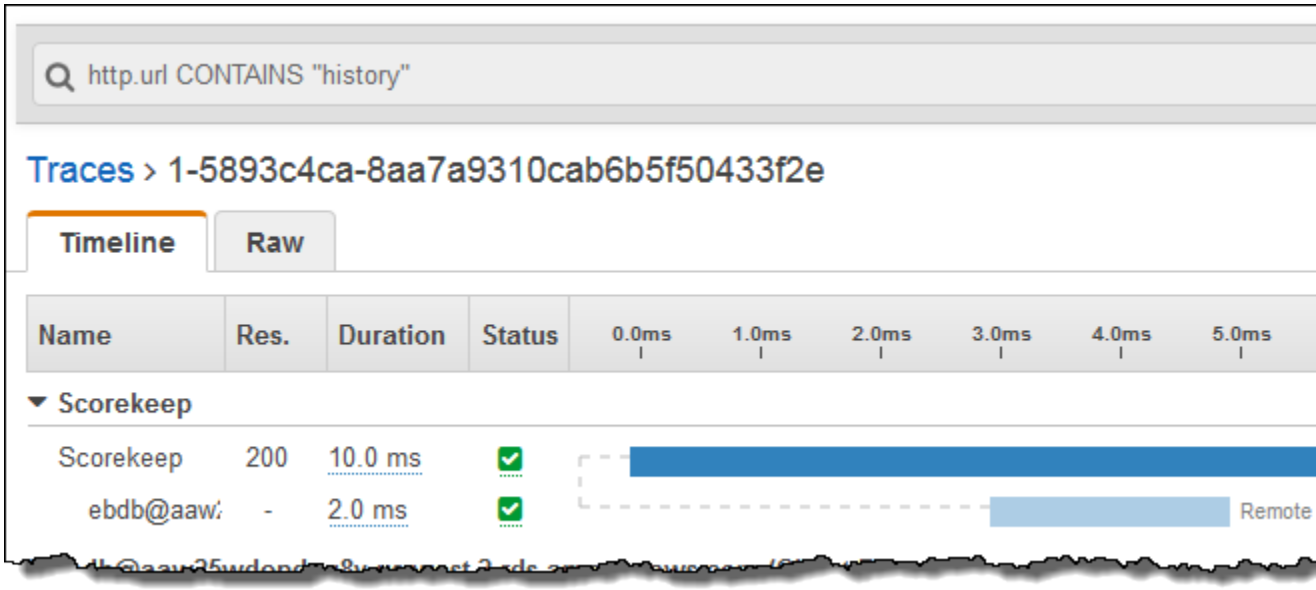
> **Note**
> See Configuring Databases with Elastic Beanstalk in the *AWS Elastic Beanstalk Developer Guide* for details on how to add a PostgreSQL database to the application environment.

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 an 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
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.



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. Any segments created by additional instrumentation become subsegments of the request-level segment that provides information about the HTTP request and response, including 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 request is forwarded to your application by a load balancer or other intermediary, The client IP in the segment is taken from the `X-Forwarded-For` header in the request instead of the source IP in the IP packet. The client IP recorded for a forwarded request can be forged so should not be trusted.

When a request is forwarded, the X-Ray SDK for Java 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 is taken from the `X-Forwarded-For` header in the HTTP request.

Sections

Adding a Tracing Filter to your Application

For Tomcat, add a `<filter>` to your project's `web.xml` file. Use the `fixedName` parameter to specify a service name (p. 56) 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>
```

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");
  }
}
```

The servlet filter 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).

Configuring a Segment Naming Strategy

The X-Ray SDK can name segments after the host name in the HTTP request header, but 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 name to use for all segments, or configure a dynamic naming strategy. A dynamic naming strategy allows the SDK to use the host name for names that match an expected pattern, and apply a default name to names that don't.

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. 55). This has the same effect as creating a FixedSegmentNamingStrategy and passing it to `AWSXRayServletFilter` constructor.

A dynamic naming strategy defines a pattern that host names should match, and a default name to use if the host name 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. 62). 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 `aws-sdk-instrumentor` submodules (p. 43) 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

```
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();
...
```

You can wrap any AWS SDK client to trace calls made using that client. For all services, you will 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 instrument a 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, along 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",
  }
}
```

Calls to the following services create additional nodes in the service map when you access named resources.

- **Amazon DynamoDB** – table name
- **Amazon Simple Storage Service** – bucket and key name
- **Amazon Simple Queue Service** – queue name

Calls to these services that don't target specific resources create a generic node for the service.

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`

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`

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;");
```

You can declare the Tomcat JDBC Data Source library 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

A segment is a JSON document that records the work that your application does to serve a single request. The AWSXRayServletFilter (p. 54) creates segments for HTTP requests and adds details about the request and response, including information from headers in the request, the time that the request was received, and the time that the response was sent.

Further instrumentation generates *subsegments*. Instrumented AWS SDK clients, HTTP clients, and JDBC clients add subsegments to the segment document with details of downstream calls made by the servlet or any functions that the servlet calls.

You can create subsegments manually to organize downstream calls into groups. For example, you can create a custom subsegment for a function that makes several calls to DynamoDB.

Example src/main/java/scorekeep/GameModel.java

```
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

```
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"
    }
  },
```

The X-Ray SDK for Node.js

The X-Ray SDK for Node.js is a library for Express framework Node.js 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 or HTTP clients.

If you use Express, start by adding the SDK as middleware (p. 67) on your application server to trace incoming requests. The middleware creates a segment 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 middleware also automatically captures exceptions that your application throws while the segment is open.

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. 65) to adjust the sampling behavior or to record information about the AWS compute resources on which your application runs.

Next, use the X-Ray SDK for Node.js to instrument your AWS SDK for JavaScript in Node.js clients (p. 68). 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. 69) to record information about outgoing HTTP calls. For SQL clients, use the capture method for your database type (p. 70).

While a segment is open, you can also create subsegments (p. 70) for any function in your application, and add debug information to segments and subsegments in annotations and metadata. X-Ray indexes annotations for use with expression filters (p. 18).

> **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.

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:

- `cls` – 0.1.5
- `continuation-local-storage` – 3.2.0
- `pkginfo` – 0.4.0
- `underscore` – 1.8.3

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.
<listitem>
`aws-sdk` – 2.7.15
</listitem>

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
nodejs-xray@0.0.0 ~/nodejs-xray
### aws-xray-sdk@1.0.4-beta
  ### continuation-local-storage@3.2.0
  # ### async-listener@0.6.3
  # # ### shimmer@1.0.0
  # ### emitter-listener@1.0.1
  ### moment@2.17.1
  ### pkginfo@0.4.0
  ### semver@5.3.0
  ### underscore@1.8.3
  ### winston@2.3.1
    ### async@1.0.0
    ### colors@1.0.3
    ### cycle@1.0.3
    ### eyes@0.1.8
    ### isstream@0.1.2
    ### stack-trace@0.0.9
```

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
nodejs-xray@0.0.0 ~/nodejs-xray
### aws-xray-sdk@1.0.4-beta
```

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 the `plugins` parameter to use a plugin that adds data about the service hosting your application.

Plugins

- `EC2` – Adds the instance ID.
- `ECS` – Adds the cluster ID.
- `ElasticBeanstalk` – Adds the environment ID.

To use a plugin, configure the X-Ray SDK for Node.js client by using the `config` method.

Example app.js

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.config([AWSXRay.plugins.EC2]);
```

Sampling Rules

The SDK has a default sampling strategy that determines which requests get traced. By default, the SDK traces the first request each second, and 5 percent of any additional requests. You can customize the SDK's sampling behavior by applying rules defined in a local file.

Example sampling-rules.json

```
{
  "rules": {
    "move": {
      "id": 1,
      "service_name": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    "base": {
      "id": 2,
      "service_name": "*",
      "http_method": "*",
      "url_path": "*",
      "fixed_target": 1,
      "rate": 0.1
    }
  }
}
```

This example defines two rules. The first rule applies a five-percent sampling rate with no minimum number of requests to trace to requests with paths under `/api/move/`. The second overrides the

default sampling rule with a rule that traces the first request each second and 10 percent of additional requests.

Tell the X-Ray SDK for Node.js to load sampling rules from a file with `setSamplingRules`.

Example app.js - sampling rules

```
var AWSXRay = require('aws-xray-sdk');
AWSXRay.setSamplingRules('sampling-rules.json');
```

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 application serves 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 framework. When you add the X-Ray middleware to your application, the X-Ray SDK for Node.js creates a segment for each sampled request. Any segments created by additional instrumentation become subsegments of the request-level segment. The request-level segment provides information about the HTTP request and response including timing, method, and disposition of the request.

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

To use the middleware, initialize the SDK client and use the middleware returned by the `express.openSegment` function before you define your routes.

Example app.js

```
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.

Segments generated by the middleware include 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).

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. 70). 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`.

```
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

```
    var AWSXRay = require('aws-xray-sdk');
...
    var ddb = AWSXRay.captureAWSClient(new AWS.DynamoDB());
```

You can wrap any AWS SDK client to trace calls made using that client. For all services, you will 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 instrument a 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, along 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",
  }
}
```

Calls to the following services create additional nodes in the service map when you access named resources.

- **Amazon DynamoDB** – table name
- **Amazon Simple Storage Service** – bucket and key name
- **Amazon Simple Queue Service** – queue name

Calls to these services that don't target specific resources create a generic node for the service.

Tracing Calls to Downstream HTTP Web Services with 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's `captureHTTPs` method to trace outgoing calls.

Example

```
var AWSXRay = require('aws-xray-sdk');
var http = require('http');

AWSXRay.captureHTTPs(http);
```

When you instrument a call to a downstream web api, the X-Ray SDK for Node.js 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 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 pg = 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.

Generating Custom Subsegments with the X-Ray SDK for Node.js

A segment is a JSON document that records the work that your application does to serve a single request. The X-Ray SDK for Node.js middleware creates segments for HTTP requests and adds

details about the request and response, including information from headers in the request, the time that the request was received, and the time that the response was sent.

Further instrumentation generates *subsegments*. Instrumented AWS SDK clients and HTTP clients add subsegments to the segment document with details of downstream calls made by the application.

You can create subsegments manually to instrument functions and organize other subsegments into groups. For example, you can create a custom subsegment for a function that makes calls to downstream services with the `captureAsync` function.

Example app.js

```
var AWSXRay = require('aws-xray-sdk');

app.use(AWSXRay.express.openSegment('MyApp'));

app.get('/', function (req, res) {
  var host = 'api.example.com';

  AWSXRay.captureAsync('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. `captureAsync` 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 `capture` 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"
    }
  },
```

The AWS X-Ray SDK for .NET

The X-Ray SDK for .NET is a library for C# .NET 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 AWS services, HTTP web APIs, and SQL databases. You can also create segments manually and add debug information in annotations and metadata.

> **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.

Download the X-Ray SDK for .NET from NuGet: nuget.org/packages/AWSXRayRecorder/

Start by adding a `TracingMessageHandler` to your web configuration (p. 75) to trace incoming requests. The message handler creates a segment to record information about each traced request that your application serves, and completes the segment when the response is sent. Within this segment you can create subsegments to trace downstream calls to other AWS services, HTTP web APIs, and SQL databases.

Next, use the X-Ray SDK for .NET to instrument your AWS SDK for .NET clients (p. 77). 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. 78) and SQL databases (p. 79). 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.

For reference documentation about the SDK's classes and methods, see the AWS X-Ray SDK for .NET API Reference.

Requirements

The X-Ray SDK for .NET requires the .NET framework and AWS SDK for .NET.

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**, choose **NuGet Package Manager**, and then choose **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.

Sections

Plugins

Use plugins to add data about the service hosting your application.

Plugins

- `EC2` – Adds the instance ID.

To use a plugin, configure the X-Ray SDK for .NET client by adding the `AWSXRayPlugins` setting.

Example Web.config - plugins

```
<configuration>
  <appSettings>
    <add key="AWSXRayPlugins" value="EC2Plugin"/>
  </appSettings>
</configuration>
```

Sampling Rules

The SDK has a default sampling strategy that determines which requests get traced. By default, the SDK traces the first request each second, and 5 percent of any additional requests. You can customize the SDK's sampling behavior by applying rules defined in a local file.

Example sampling-rules.json

```
{
  "rules": {
    "move": {
      "id": 1,
      "service_name": "*",
      "http_method": "*",
      "url_path": "/api/move/*",
      "fixed_target": 0,
      "rate": 0.05
    },
    "base": {
      "id": 2,
      "service_name": "*",
      "http_method": "*",
      "url_path": "*",
      "fixed_target": 1,
      "rate": 0.1
    }
  }
}
```

This example defines two rules. The first rule applies a five-percent sampling rate with no minimum number of requests to trace to requests with paths under `/api/move/`. The second overrides the default sampling rule with a rule that traces the first request each second and 10 percent of additional requests.

Tell the X-Ray SDK for .NET to load sampling rules from a file with the `SamplingRuleManifest` setting.

Example Web.config - sampling rules

```
<configuration>
  <appSettings>
    <add key="SamplingRuleManifest" value="sampling-rules.json"/>
  </appSettings>
</configuration>
```

Instrumenting Incoming HTTP Requests with the X-Ray SDK for .NET

To instrument requests served by your application, add a `TracingMessageHandler` to the `HttpConfiguration.MessageHandlers` collection in your web configuration.

Example WebApiConfig - Message handler

```
using System.Web.Http;
using Amazon.XRay.Recorder.Handler.Http;
using SampleEBWebApplication.Controllers;

namespace SampleEBWebApplication
{
  public static class WebApiConfig
  {
    public static void Register(HttpConfiguration config)
    {
      // Add the message handler to HttpCofiguration
      config.MessageHandlers.Add(new TracingMessageHandler());
      // Web API routes
      config.MapHttpAttributeRoutes();
      config.Routes.MapHttpRoute(
        name: "DefaultApi",
        routeTemplate: "api/{controller}/{id}",
        defaults: new { id = RouteParameter.Optional }
      );
    }
  }
}
```

When you add the X-Ray message handler to your application, the X-Ray SDK for .NET 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, including timing, method, and disposition of the request.

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

Alternatively, you can also add the tracing handler to a `global.asax` file.

Example global.asax - Message handler

```
using System.Web.Http;

namespace SampleEBWebApplication
{
  public class WebApiApplication : System.Web.HttpApplication
  {
    protected void Application_Start()
    {
      GlobalConfiguration.Configure(WebApiConfig.Register);
      GlobalConfiguration.Configuration.MessageHandlers.Add(new
 TracingMessageHandler());
    }
  }
}
```

Instrumenting Downstream Calls to AWS Services

You can instrument your AWS SDK for .NET clients by adding an event handler with `AWSXRayRecorder.Instance.AddEventHandler`.

Example SampleController.cs - DynamoDB Client Instrumentation

Initialize a DynamoDB client with the AWS SDK for Java, and then add the event handler with the default `AWSXRayRecorder`'s `AddEventHandler` method.

```
using Amazon;
using Amazon.Util;
using Amazon.DynamoDBv2;
using Amazon.DynamoDBv2.DocumentModel;
using Amazon.XRay.Recorder.Core;

namespace SampleEBWebApplication.Controllers
{
  public class SampleController : ApiController
  {
    private static readonly Lazy<AmazonDynamoDBClient> LazyDdbClient = new
 Lazy<AmazonDynamoDBClient>(() =>
    {
      var client = new AmazonDynamoDBClient(EC2InstanceMetadata.Region ??
 RegionEndpoint.USEast1);
      AWSXRayRecorder.Instance.AddEventHandler(client);
      return client;
    });
```

You can wrap any AWS SDK client to trace calls made using that client. For all services, you will 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 instrument a 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, along 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",
  }
}
```

Calls to the following services create additional nodes in the service map when you access named resources.

- **Amazon DynamoDB** – table name
- **Amazon Simple Storage Service** – bucket and key name
- **Amazon Simple Queue Service** – queue name

Calls to these services that don't target specific resources create a generic node for the service.

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 HttpClient

```
using System.Net;
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handler.Http;

private void MakeHttpRequest()
{
  HttpWebRequest request = (HttpWebRequest)WebRequest.Create("http://
names.example.com/api");
  request.GetResponseTraced();
}
```

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 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 .NET

The SDK provides a wrapper class for `System.Data.SqlClient.SqlCommand` named `TraceableSqlCommand` that you can use in place of `SqlCommand`.

Example `Controller.cs` - SQL Client Instrumentation

Initialize an SQL command with the X-Ray SDK for .NET's `TraceableSqlCommand` class.

```
using Amazon;
using Amazon.Util;
using Amazon.XRay.Recorder.Core;
using Amazon.XRay.Recorder.Handler.Sql;
  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();
    }
  }
```

Creating Additional Subsegments

You can add subsegments to request segments with `BeginSubsegment` and `EndSubsegment`. 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.

```
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"
    }
  },
```

The 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.

You can download the daemon from Amazon S3.

- **Linux (executable)** – `aws-xray-daemon-linux-1.x.zip`
- **Linux (RPM installer)** – `aws-xray-daemon-1.x.rpm`
- **Linux (DEB installer)** – `aws-xray-daemon-1.x.deb`
- **Windows (executable)** – `aws-xray-daemon-windows-process-1.x.zip`
- **Windows (service)** – `aws-xray-daemon-windows-service-1.x.zip`

Run the daemon from a command line.

```
~/Downloads$ ./xray
```

For detailed platform-specific instructions, see the following:

- **Linux (local)** – Running the X-Ray Daemon on Linux (p. 86)
- **Windows (local)** – Running the X-Ray Daemon on Windows (p. 86)
- **Elastic Beanstalk** – Running the X-Ray Daemon on AWS Elastic Beanstalk (p. 87)
- **Amazon EC2** – Running the X-Ray Daemon on Amazon EC2 (p. 90)
- **Amazon ECS** – Running the X-Ray Daemon on Amazon ECS (p. 91)

Sections

- Running the X-Ray Daemon Locally (p. 85)
- Running the X-Ray Daemon on AWS Elastic Beanstalk (p. 87)
- Running the X-Ray Daemon on Amazon EC2 (p. 90)
- Running the X-Ray Daemon on Amazon ECS (p. 91)

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
```

For more information about providing credentials to an 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.

To create an instance profile for use with X-Ray on Amazon EC2

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

   - **AWSXrayWriteOnlyAccess** – Gives the X-Ray daemon permission to upload trace data.
   - **AmazonS3ReadOnlyAccess** – 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**.

To add X-Ray write permissions to an IAM user, group, or role

1. Open the IAM console.
2. Open the role associated with your instance profile, your IAM user, or your IAM user's group.
3. Under **Permissions**, add the following managed policies.

   - **AWSXrayWriteOnlyAccess** – Gives the X-Ray daemon permission to upload trace data.
   - **AmazonS3ReadOnlyAccess** – Gives the instance or IAM user permission to download the X-Ray daemon from Amazon S3.

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 1.0.0
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)
```

Configuring the Daemon

You can use command line options to customize the daemon's behavior.

Command line options

- `-b`, `--bind` – Bind the daemon to a different port.

  ```
  --bind "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`, `--memory-limit` – Change the amount of memory (in MiB) that the daemon can use.

```
--memory-limit 150
```

Default – 5% 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"
```

- `-v`, `--version` – Show AWS X-Ray daemon version.
- `-h`, `--help` – Show the help screen.

You can load a YAML format configuration file with the `--config` option.

Configuration file options

- `LocalMode` – Set to `true` to skip checking for EC2 instance metadata.
- `Logging` – Configure logging behavior.
  - `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.
- `Processor` – Configure the daemon process.
  - `Region` – Specify a region to send trace data to that region instead of the current region.
- `RoleARN` – Assume the specified IAM role to upload segments to a different account.
- `Socket` – Configure the daemon's binding.
  - `UDPAddress` – Change the port on which the daemon listens.

Example xray-daemon.yaml

```
Socket:
  UDPAddress: "127.0.0.1:3000"
Processor:
  Region: "us-east-2"
Logging:
  LogLevel: "warn"
  LogPath: "/var/log/xray-daemon.log"
LocalMode: true
RoleARN: "arn:aws:iam::123456789012:role/xray-cross-account"
```

Pass the configuration file to the daemon by using the `-c` option.

```
~$ ./xray -c ~/xray-daemon.yaml
```

Running the X-Ray Daemon Locally

You can run the daemon locally for development and testing.

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. 83).

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 Daemon (p. 84).

Running the X-Ray Daemon on Linux

You can run the daemon executable from the command line, as follows.

```
~/xray-daemon$ ./xray
```

To run the daemon in the background, use `&`.

```
~/xray-daemon$ ./xray &
```

Terminate a daemon process running in the background with `pkill`.

```
~$ pkill xray
```

Running the X-Ray Daemon on Windows

You can run the daemon executable from the command line.

```
> .\xray_windows.exe
```

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-1.x.zip"
$zipPath = "$currentLocation\$zipFileName"
$destPath = "$currentLocation\aws-xray-daemon"
$daemonPath = "$destPath\xray.exe"
$daemonLogPath = "C:\inetpub\wwwroot\xray-daemon.log"
$url = "https://s3.amazonaws.com/aws-xray-assets.us-east-1/xray-daemon/aws-
xray-daemon-windows-service-1.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 AWS Elastic Beanstalk

You can run the X-Ray daemon on your Elastic Beanstalk environment's EC2 instances to relay trace data from your application to AWS X-Ray.

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. 83).

AWS 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 checking an option in the Elastic Beanstalk console.

The daemon is installed on the instance and runs as a service when you enable the configuration option.

The version included on Elastic Beanstalk platforms may not be the latest version. See the Elastic Beanstalk release notes to find out the version of the daemon that is available for the platform configuration that you use.

Using Elastic Beanstalk's 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:\Progam Files\Amazon\XRay\logs\xray-service.log`

On Linux Platforms

- **Configuration file** – `/etc/amazon/xray/cfg.yaml`
- **Logs** – `/var/log/xray/xray.log`

You can tell Elastic Beanstalk to pull the daemon logs on demand by adding a file to the log tasks directory 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 Advanced Environment Customization with Configuration Files (.ebextensions) 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.amazonaws.com/aws-xray-assets.us-east-1/xray-
daemon/aws-xray-daemon-1.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-scirpt:
    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-1.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-east-1.amazonaws.com/aws-xray-
assets.us-east-1/xray-daemon/aws-xray-daemon-windows-service-1.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 adds the daemon's log file to Elastic Beanstalk's tail logs task, to include it 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. 83).

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.amazonaws.com/aws-xray-assets.us-east-1/xray-daemon/aws-xray-
daemon-1.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-1.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-east-1.amazonaws.com/aws-xray-assets.us-
east-1/xray-daemon/aws-xray-daemon-windows-service-1.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

On Amazon ECS, create a Docker image that runs the daemon, upload it to a Docker image repository, and then deploy it to your Amazon ECS cluster.

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. 83).

Use a Dockerfile file to create a Docker image that runs the daemon.

Example Dockerfile

```
FROM ubuntu:12.04
COPY xray /usr/bin/xray
CMD xray --log-file /var/log/xray-daemon.log &
```

Download the X-Ray daemon (p. 82) Linux executable into the same folder as your Dockerfile and build it to create an image.

Integrating AWS X-Ray with AWS Services

Other AWS services provide integration with AWS X-Ray by adding trace IDs to requests, making sampling decisions, or 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.

Elastic Load Balancing

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 2nd, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal.
- A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.

Amazon API Gateway

Amazon API Gateway gateways 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 2nd, 2016 PST in epoch time is `1480615200` seconds, or `58406520` in hexadecimal.
- A 96-bit identifier for the trace, globally unique, in **24 hexadecimal digits**.

API Gateway does not propagate X-Ray trace ID and sampling headers. If your gateway is downstream of other services in your application, traces will terminate at the gateway. If the gateway chooses to sample the request, it will continue with a different trace ID.

Amazon Elastic Compute Cloud

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. 90) 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. 83).

AWS Elastic Beanstalk

AWS Elastic Beanstalk platforms include the X-Ray daemon. You can run the daemon (p. 87) 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.

For more information, see Configuring AWS X-Ray Debugging in the AWS Elastic Beanstalk Developer Guide.