

Quarkus - Amazon SQS Client

Amazon Simple Queue Service (SQS) is a fully managed message queuing service. Using SQS, you can send, store, and receive messages between software components at any volume, without losing messages or requiring other services to be available. SQS offers two types of message queues. Standard queues offer maximum throughput, best-effort ordering and at-least-once delivery. SQS FIFO queues are designed to guarantee that messages are processed exactly once, in the exact order that they were sent.

You can find more information about SQS at [the Amazon SQS website](#).



The SQS extension is based on [AWS Java SDK 2.x](#). It's a major rewrite of the 1.x code base that offers two programming models (Blocking & Async).

This technology is considered preview.



In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

The Quarkus extension supports two programming models:

- Blocking access using URL Connection HTTP client (by default) or the Apache HTTP Client
- [Asynchronous programming](#) based on JDK's `CompletableFuture` objects and the Netty HTTP client.

In this guide, we see how you can get your REST services to use SQS locally and on AWS.

Prerequisites

To complete this guide, you need:

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- an IDE
- Apache Maven 3.6.3
- An AWS Account to access the SQS service
- Docker for your system to run SQS locally for testing purposes

Set up SQS locally

The easiest way to start working with SQS is to run a local instance as a container.

```
docker run --rm --name local-sqs 8010:4576 -e SERVICES=sqs -e  
START_WEB=0 -d localstack/localstack:0.11.1
```

This starts a SQS instance that is accessible on port **8010**.

Create an AWS profile for your local instance using AWS CLI:

```
$ aws configure --profile localstack  
AWS Access Key ID [None]: test-key  
AWS Secret Access Key [None]: test-secret  
Default region name [None]: us-east-1  
Default output format [None]:
```

Create a SQS queue

Create a SQS queue using AWS CLI and store in **QUEUE_URL** environment variable.

```
QUEUE_URL=`aws sqs create-queue --queue-name=ColliderQueue  
--profile localstack --endpoint-url=http://localhost:8010`
```

Or, if you want to use your SQS queue on your AWS account create a queue using your default profile

```
QUEUE_URL=`aws sqs create-queue --queue-name=ColliderQueue`
```

Solution

The application built here allows shooting an elementary particles (quarks) into a **ColliderQueue** queue of the AWS SQS. Additionally, we create a resource that allows receiving those quarks from the **ColliderQueue** queue in the order they were sent.

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the **amazon-sqs-quickstart** directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.5.0.CR1:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=amazon-sqs-quickstart \
  -DclassName="org.acme.sqs.QuarksCannonSyncResource" \
  -Dpath="/sync-cannon" \
  -Dextensions="resteasy-jsonb,amazon-sqs,resteasy-mutiny"
cd amazon-sqs-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS, Mutiny and Amazon SQS Client extensions. After this, the `amazon-sqs` extension has been added to your `pom.xml` as well as the Mutiny support for RESTEasy.

Creating JSON REST service

In this example, we will create an application that sends quarks via the queue. The example application will demonstrate the two programming models supported by the extension.

First, let's create the `Quark` bean as follows:

```

package org.acme.sqs.model;

import io.quarkus.runtime.annotations.RegisterForReflection;
import java.util.Objects;

@RegisterForReflection
public class Quark {

    private String flavor;
    private String spin;

    public Quark() {
    }

    public String getFlavor() {
        return flavor;
    }

    public void setFlavor(String flavor) {
        this.flavor = flavor;
    }

    public String getSpin() {
        return spin;
    }

    public void setSpin(String spin) {
        this.spin = spin;
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Quark)) {
            return false;
        }

        Quark other = (Quark) obj;

        return Objects.equals(other.flavor, this.flavor);
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.flavor);
    }
}

```

Then, create a `org.acme.sqs.QuarksCannonSyncResource` that will provide an API to shoot quarks into the SQS queue using the synchronous client.

```

package org.acme.sqs;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectWriter;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.acme.sqs.model.Quark;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sqs.SqsClient;
import software.amazon.awssdk.services.sqs.model.SendMessageResponse;

@Path("/sync/cannon")
@Produces(MediaType.TEXT_PLAIN)
public class QuarksCannonSyncResource {

    private static final Logger LOGGER = Logger.getLogger(
        QuarksCannonSyncResource.class);

    @Inject
    SqsClient sqs;

    @ConfigProperty(name = "queue.url")
    String queueUrl;

    static ObjectWriter QUARK_WRITER = new ObjectMapper().
writerFor(Quark.class);

    @POST
    @Path("/shoot")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response sendMessage(Quark quark) throws Exception {
        String message = QUARK_WRITER.writeValueAsString(quark);
        SendMessageResponse response = sqs.sendMessage(m -> m
.queueUrl(queueUrl).messageBody(message));
        LOGGER.infov("Fired Quark[{0}, {1}]", quark.getFlavor(),
quark.getSpin());
        return Response.ok().entity(response.messageId()).build();
    }
}

```

Because of the fact messages sent to the queue must be a **String**, we're using Jackson's

`ObjectWriter` in order to serialize our `Quark` objects into a `String`.

Now, create the `org.acme.QuarksShieldSyncResource` REST resources that provides an endpoint to read the messages from the `ColliderQueue` queue.

```
package org.acme.sqs;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectReader;
import java.util.List;
import java.util.stream.Collectors;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.acme.sqs.model.Quark;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sqs.SqsClient;
import software.amazon.awssdk.services.sqs.model.Message;

@Path("/sync/shield")
public class QuarksShieldSyncResource {

    private static final Logger LOGGER = Logger.getLogger(
        QuarksShieldSyncResource.class);

    @Inject
    SqsClient sqs;

    @ConfigProperty(name = "queue.url")
    String queueUrl;

    static ObjectReader QUARK_READER = new ObjectMapper().
        readerFor(Quark.class);

    @GET
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public List<Quark> receive() {
        List<Message> messages = sqs.receiveMessage(m -> m
            .numberOfMessages(10).queueUrl(queueUrl)).messages();

        return messages.stream()
            .map(Message::body)
            .map(this::toQuark)
            .collect(Collectors.toList());
    }
}
```

```

    }

    private Quark toQuark(String message) {
        Quark quark = null;
        try {
            quark = QUARK_READER.readValue(message);
        } catch (Exception e) {
            LOGGER.error("Error decoding message", e);
            throw new RuntimeException(e);
        }
        return quark;
    }
}

```

We are using here a Jackson's `ObjectReader` in order to deserialize queue messages into our `Quark` POJOs.

Configuring SQS clients

Both SQS clients (sync and async) are configurable via the `application.properties` file that can be provided in the `src/main/resources` directory. Additionally, you need to add to the classpath a proper implementation of the sync client. By default the extension uses the URL connection HTTP client, so you need to add a URL connection client dependency to the `pom.xml` file:

```

<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>url-connection-client</artifactId>
</dependency>

```

If you want to use Apache HTTP client instead, configure it as follows:

```
quarkus.sqs.sync-client.type=apache
```

And add the following dependency to the application `pom.xml`:

```

<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>apache-client</artifactId>
</dependency>

```

If you're going to use a local SQS instance, configure it as follows:

```
quarkus.sqs.endpoint-override=http://localhost:8010

quarkus.sqs.aws.region=us-east-1
quarkus.sqs.aws.credentials.type=static
quarkus.sqs.aws.credentials.static-provider.access-key-id=test-key
quarkus.sqs.aws.credentials.static-provider.secret-access-key=test-secret
```

- `quarkus.sqs.aws.region` - It's required by the client, but since you're using a local SQS instance use `us-east-1` as it's a default region of localstack's SQS.
- `quarkus.sqs.aws.credentials.type` - Set `static` credentials provider with any values for `access-key-id` and `secret-access-key`
- `quarkus.sqs.endpoint-override` - Override the SQS client to use a local instance instead of an AWS service

If you want to work with an AWS account, you can simply remove or comment out all SQS related properties. By default, the SQS client extension will use the `default` credentials provider chain that looks for credentials in this order: - Java System Properties - `aws.accessKeyId` and `aws.secretKey` * Environment Variables - `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` * Credential profiles file at the default location (`~/.aws/credentials`) shared by all AWS SDKs and the AWS CLI * Credentials delivered through the Amazon EC2 container service if the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` environment variable is set and the security manager has permission to access the variable, * Instance profile credentials delivered through the Amazon EC2 metadata service

And the region from your AWS CLI profile will be used.

Next steps

Packaging

Packaging your application is as simple as `./mvnw clean package`. It can be run with `java -Dqueue.url=$QUEUE_URL -jar target/amazon-sqs-quickstart-1.0-SNAPSHOT-runner.jar`.

With GraalVM installed, you can also create a native executable binary: `./mvnw clean package -Dnative`. Depending on your system, that will take some time.

Going asynchronous

Thanks to the AWS SDK v2.x used by the Quarkus extension, you can use the asynchronous programming model out of the box.

Create a `org.acme.sqs.QuarksCannonAsyncResource` REST resource that will be similar to our `QuarksCannonSyncResource` but using an asynchronous programming model.


```

package org.acme.sqs;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectWriter;
import io.smallrye.mutiny.Uni;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.acme.sqs.model.Quark;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sqs.SqsAsyncClient;
import software.amazon.awssdk.services.sqs.model.SendMessageResponse;

@Path("/async/cannon")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class QuarksCannonAsyncResource {

    private static final Logger LOGGER = Logger.getLogger
(QuarksCannonAsyncResource.class);

    @Inject
    SqsAsyncClient sqs;

    @ConfigProperty(name = "queue.url")
    String queueUrl;

    static ObjectWriter QUARK_WRITER = new ObjectMapper().
writerFor(Quark.class);

    @POST
    @Path("/shoot")
    @Consumes(MediaType.APPLICATION_JSON)
    public Uni<Response> sendMessage(Quark quark) throws Exception
    {
        String message = QUARK_WRITER.writeValueAsString(quark);
        return Uni.createFrom()
            .completionStage(sqs.sendMessage(m -> m.queueUrl
(queueUrl).messageBody(message)))
            .onItem().invoke(item -> LOGGER.infov("Fired Quark[{0},
{1}]]", quark.getFlavor(), quark.getSpin()))
            .onItem().apply(SendMessageResponse::messageId)
            .onItem().apply(id -> Response.ok().entity(id).build())
    }

```

```
);
    }
}
```

We create `Uni` instances from the `CompletionStage` objects returned by the asynchronous SQS client, and then transform the emitted item.

And the corresponding async receiver of the queue messages `org.acme.sqs.QuarksShieldAsyncResource`

```
package org.acme.sqs;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectReader;
import io.smallrye.mutiny.Uni;
import java.util.List;
import java.util.stream.Collectors;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import org.acme.sqs.model.Quark;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sqs.SqsAsyncClient;
import software.amazon.awssdk.services.sqs.model.Message;
import software.amazon.awssdk.services.sqs.model.ReceiveMessageResponse;

@Path("/async/shield")
public class QuarksShieldAsyncResource {

    private static final Logger LOGGER = Logger.getLogger(
        QuarksShieldAsyncResource.class);

    @Inject
    SqsAsyncClient sqs;

    @ConfigProperty(name = "queue.url")
    String queueUrl;

    static ObjectReader QUARK_READER = new ObjectMapper().
        readerFor(Quark.class);

    @GET
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
```

```

public Uni<List<Quark>> receive() {
    return Uni.createFrom()
        .completionStage(sqs.receiveMessage(m -> m
            .maxNumberOfMessages(10).queueUrl(queueUrl)))
        .onItem().apply(ReceiveMessageResponse::messages)
        .onItem().apply(m -> m.stream().map(Message::body).map
            (this::toQuark).collect(Collectors.toList()));
}

private Quark toQuark(String message) {
    Quark quark = null;
    try {
        quark = QUARK_READER.readValue(message);
    } catch (Exception e) {
        LOGGER.error("Error decoding message", e);
        throw new RuntimeException(e);
    }
    return quark;
}
}

```


And we need to add the Netty HTTP client dependency to the `pom.xml`:

```

<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>netty-nio-client</artifactId>
</dependency>

```

Configuration Reference

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.sqs.interceptors</code> List of execution interceptors that will have access to read and modify the request and response objects as they are processed by the AWS SDK. The list should consists of class names which implements <code>software.amazon.awssdk.core.interceptor.ExecutionInterceptor</code> or interface.	list of class name	
 <code>quarkus.sqs.sync-client.type</code> Type of the sync HTTP client implementation	url, apache	url

AWS SDK client configurations	Type	Default
<p><code>quarkus.sqs.endpoint-override</code></p> <p>The endpoint URI with which the SDK should communicate. If not specified, an appropriate endpoint to be used for the given service and region.</p>	URI	
<p><code>quarkus.sqs.api-call-timeout</code></p> <p>The amount of time to allow the client to complete the execution of an API call. This timeout covers the entire client execution except for marshalling. This includes request handler execution, all HTTP requests including retries, unmarshalling, etc. This value should always be positive, if present.</p>	Duration ?	
<p><code>quarkus.sqs.api-call-attempt-timeout</code></p> <p>The amount of time to wait for the HTTP request to complete before giving up and timing out. This value should always be positive, if present.</p>	Duration ?	
AWS services configurations	Type	Default
<p><code>quarkus.sqs.aws.region</code></p> <p>An Amazon Web Services region that hosts the given service.</p> <p>It overrides region provider chain with static value of region with which the service client should communicate.</p> <p>If not set, region is retrieved via the default providers chain in the following order:</p> <ul style="list-style-type: none"> • <code>aws.region</code> system property • <code>region</code> property from the profile file • Instance profile file <p>See <code>software.amazon.awssdk.regions.Region</code> for available regions.</p>	Region	

quarkus.sqs.aws.credentials.type

Configure the credentials provider that should be used to authenticate with AWS.

Available values:

- **default** - the provider will attempt to identify the credentials automatically using the following checks:
 - Java System Properties - `aws.accessKeyId` and `aws.secretKey`
 - Environment Variables - `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
 - Credential profiles file at the default location (`~/.aws/credentials`) shared by all AWS SDKs and the AWS CLI
 - Credentials delivered through the Amazon EC2 container service if `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` environment variable is set and security manager has permission to access the variable.
 - Instance profile credentials delivered through the Amazon EC2 metadata service
- **static** - the provider that uses the access key and secret access key specified in the `static-provider` section of the config.
- **system-property** - it loads credentials from the `aws.accessKeyId`, `aws.secretAccessKey` and `aws.sessionToken` system properties.
- **env-variable** - it loads credentials from the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` and `AWS_SESSION_TOKEN` environment variables.
- **profile** - credentials are based on AWS configuration profiles. This loads credentials from a [profile file](#), allowing you to share multiple sets of AWS security credentials between different tools like the AWS SDK for Java and the AWS CLI.
- **container** - It loads credentials from a local metadata service. Containers currently supported by the AWS SDK are **Amazon Elastic Container Service (ECS)** and **AWS Greengrass**
- **instance-profile** - It loads credentials from the Amazon EC2 Instance Metadata Service.
- **process** - Credentials are loaded from an external process. This is used to support the `credential_process` setting in the profile credentials file. See [Sourcing Credentials From External Processes](#) for more information.
- **anonymous** - It always returns anonymous AWS credentials. Anonymous AWS credentials result in un-authenticated requests and will fail unless the resource or API's policy has been configured to specifically allow anonymous access.

default,
static,
system-
prop-
erty,
env-
variab
le,
profil
e,
contai
ner,
instan
ce-
profil
e,
proces
s,
anonym
ous

default

Default credentials provider configuration	Type	Default
<code>quarkus.sqs.aws.credentials.default-provider.async-credential-update-enabled</code> Whether this provider should fetch credentials asynchronously in the background. If this is <code>true</code> , threads are less likely to block, but additional resources are used to maintain the provider.	boolean	<code>false</code>
<code>quarkus.sqs.aws.credentials.default-provider.reuse-last-provider-enabled</code> Whether the provider should reuse the last successful credentials provider in the chain. Reusing the last successful credentials provider will typically return credentials faster than searching through the chain.	boolean	<code>true</code>
Static credentials provider configuration	Type	Default
<code>quarkus.sqs.aws.credentials.static-provider.access-key-id</code> AWS Access key id	string	
<code>quarkus.sqs.aws.credentials.static-provider.secret-access-key</code> AWS Secret access key	string	
AWS Profile credentials provider configuration	Type	Default
<code>quarkus.sqs.aws.credentials.profile-provider.profile-name</code> The name of the profile that should be used by this credentials provider. If not specified, the value in <code>AWS_PROFILE</code> environment variable or <code>aws.profile</code> system property is used and defaults to <code>default</code> name.	string	
Process credentials provider configuration	Type	Default
<code>quarkus.sqs.aws.credentials.process-provider.async-credential-update-enabled</code> Whether the provider should fetch credentials asynchronously in the background. If this is true, threads are less likely to block when credentials are loaded, but additional resources are used to maintain the provider.	boolean	<code>false</code>
<code>quarkus.sqs.aws.credentials.process-provider.credential-refresh-threshold</code> The amount of time between when the credentials expire and when the credentials should start to be refreshed. This allows the credentials to be refreshed before they are reported to expire.	Duration 	<code>15S</code>

<code>quarkus.sqs.aws.credentials.process-provider.process-output-limit</code>	Memory Size ?	1024
The maximum size of the output that can be returned by the external process before an exception is raised.		
<code>quarkus.sqs.aws.credentials.process-provider.command</code>	string	
The command that should be executed to retrieve credentials.		
Sync HTTP transport configurations	Type	Default
<code>quarkus.sqs.sync-client.connection-timeout</code>	Duration ?	2S
The maximum amount of time to establish a connection before timing out.		
<code>quarkus.sqs.sync-client.socket-timeout</code>	Duration ?	30S
The amount of time to wait for data to be transferred over an established, open connection before the connection is timed out.		
Apache HTTP client specific configurations	Type	Default
<code>quarkus.sqs.sync-client.apache.connection-acquisition-timeout</code>	Duration ?	10S
The amount of time to wait when acquiring a connection from the pool before giving up and timing out.		
<code>quarkus.sqs.sync-client.apache.connection-max-idle-time</code>	Duration ?	60S
The maximum amount of time that a connection should be allowed to remain open while idle.		
<code>quarkus.sqs.sync-client.apache.connection-time-to-live</code>	Duration ?	
The maximum amount of time that a connection should be allowed to remain open, regardless of usage frequency.		
<code>quarkus.sqs.sync-client.apache.max-connections</code>	int	50
The maximum number of connections allowed in the connection pool. Each built HTTP client has its own private connection pool.		
<code>quarkus.sqs.sync-client.apache.expect-continue-enabled</code>	boolean	true
Whether the client should send an HTTP expect-continue handshake before each request.		

<code>quarkus.sqs.sync-client.apache.use-idle-connection-reaper</code> Whether the idle connections in the connection pool should be closed asynchronously. When enabled, connections left idling for longer than <code>quarkus..sync-client.connection-max-idle-time</code> will be closed. This will not close connections currently in use.	boolean	<code>true</code>
<code>quarkus.sqs.sync-client.apache.proxy.enabled</code> Enable HTTP proxy	boolean	<code>false</code>
<code>quarkus.sqs.sync-client.apache.proxy.endpoint</code> The endpoint of the proxy server that the SDK should connect through. Currently, the endpoint is limited to a host and port. Any other URI components will result in an exception being raised.	URI	
<code>quarkus.sqs.sync-client.apache.proxy.username</code> The username to use when connecting through a proxy.	string	
<code>quarkus.sqs.sync-client.apache.proxy.password</code> The password to use when connecting through a proxy.	string	
<code>quarkus.sqs.sync-client.apache.proxy.ntlm-domain</code> For NTLM proxies - the Windows domain name to use when authenticating with the proxy.	string	
<code>quarkus.sqs.sync-client.apache.proxy.ntlm-workstation</code> For NTLM proxies - the Windows workstation name to use when authenticating with the proxy.	string	
<code>quarkus.sqs.sync-client.apache.proxy.preemptive-basic-authentication-enabled</code> Whether to attempt to authenticate preemptively against the proxy server using basic authentication.	boolean	
<code>quarkus.sqs.sync-client.apache.proxy.non-proxy-hosts</code> The hosts that the client is allowed to access without going through the proxy.	list of string	

<code>quarkus.sqs.sync-client.apache.tls-managers-provider.type</code> TLS managers provider type. Available providers: <ul style="list-style-type: none"> • none - Use this provider if you don't want the client to present any certificates to the remote TLS host. • system-property - Provider checks the standard <code>javax.net.ssl.keyStore</code>, <code>javax.net.ssl.keyStorePassword</code>, and <code>javax.net.ssl.keyStoreType</code> properties defined by the JSSE. • file-store - Provider that loads a the key store from a file. 	none, system- -prop erty, file- store	system- -prop erty
<code>quarkus.sqs.sync-client.apache.tls-managers-provider.file-store.path</code> Path to the key store.	path	
<code>quarkus.sqs.sync-client.apache.tls-managers-provider.file-store.type</code> Key store type. See the KeyStore section in the Java Cryptography Architecture Standard Algorithm Name Documentation for information about standard keystore types.	string	
<code>quarkus.sqs.sync-client.apache.tls-managers-provider.file-store.password</code> Key store password	string	
Netty HTTP transport configurations	Type	Default
<code>quarkus.sqs.async-client.max-concurrency</code> The maximum number of allowed concurrent requests. For HTTP/1.1 this is the same as max connections. For HTTP/2 the number of connections that will be used depends on the max streams allowed per connection.	int	50
<code>quarkus.sqs.async-client.max-pending-connection-acquires</code> The maximum number of pending acquires allowed. Once this exceeds, acquire tries will be failed.	int	10000
<code>quarkus.sqs.async-client.read-timeout</code> The amount of time to wait for a read on a socket before an exception is thrown. Specify 0 to disable.	Duration ?	30S

<code>quarkus.sqs.async-client.write-timeout</code>	Duration ?	30S
The amount of time to wait for a write on a socket before an exception is thrown. Specify 0 to disable.		
<code>quarkus.sqs.async-client.connection-timeout</code>	Duration ?	10S
The amount of time to wait when initially establishing a connection before giving up and timing out.		
<code>quarkus.sqs.async-client.connection-acquisition-timeout</code>	Duration ?	2S
The amount of time to wait when acquiring a connection from the pool before giving up and timing out.		
<code>quarkus.sqs.async-client.connection-time-to-live</code>	Duration ?	
The maximum amount of time that a connection should be allowed to remain open, regardless of usage frequency.		
<code>quarkus.sqs.async-client.connection-max-idle-time</code>	Duration ?	60S
The maximum amount of time that a connection should be allowed to remain open while idle. Currently has no effect if <code>quarkus..async-client.use-idle-connection-reaper</code> is false.		
<code>quarkus.sqs.async-client.use-idle-connection-reaper</code>	boolean	true
Whether the idle connections in the connection pool should be closed. When enabled, connections left idling for longer than <code>quarkus..async-client.connection-max-idle-time</code> will be closed. This will not close connections currently in use.		
<code>quarkus.sqs.async-client.protocol</code>	http1-1, http2	http1-1
The HTTP protocol to use.		
<code>quarkus.sqs.async-client.ssl-provider</code>	jdk, openssl, openssl- refcnt	
The SSL Provider to be used in the Netty client. Default is <code>OPENSSL</code> if available, <code>JDK</code> otherwise.		
<code>quarkus.sqs.async-client.http2.max-streams</code>	long	4294967295
The maximum number of concurrent streams for an HTTP/2 connection. This setting is only respected when the HTTP/2 protocol is used.		

<code>quarkus.sqs.async-client.http2.initial-window-size</code> The initial window size for an HTTP/2 stream. This setting is only respected when the HTTP/2 protocol is used.	int	1048576
<code>quarkus.sqs.async-client.proxy.enabled</code> Enable HTTP proxy.	boolean	false
<code>quarkus.sqs.async-client.proxy.endpoint</code> The endpoint of the proxy server that the SDK should connect through. Currently, the endpoint is limited to a host and port. Any other URI components will result in an exception being raised.	URI	
<code>quarkus.sqs.async-client.proxy.non-proxy-hosts</code> The hosts that the client is allowed to access without going through the proxy.	list of string	
<code>quarkus.sqs.async-client.tls-managers-provider.type</code> TLS managers provider type. Available providers: <ul style="list-style-type: none"> • none - Use this provider if you don't want the client to present any certificates to the remote TLS host. • system-property - Provider checks the standard <code>javax.net.ssl.keyStore</code>, <code>javax.net.ssl.keyStorePassword</code>, and <code>javax.net.ssl.keyStoreType</code> properties defined by the JSSE. • file-store - Provider that loads a the key store from a file. 	none, system-property, file-store	system-property
<code>quarkus.sqs.async-client.tls-managers-provider.file-store.path</code> Path to the key store.	path	
<code>quarkus.sqs.async-client.tls-managers-provider.file-store.type</code> Key store type. See the KeyStore section in the Java Cryptography Architecture Standard Algorithm Name Documentation for information about standard keystore types.	string	

<code>quarkus.sqs.async-client.tls-managers-provider.file-store.password</code> Key store password	string	
<code>quarkus.sqs.async-client.event-loop.override</code> Enable the custom configuration of the Netty event loop group.	boolean	<code>false</code>
<code>quarkus.sqs.async-client.event-loop.number-of-threads</code> Number of threads to use for the event loop group. If not set, the default Netty thread count is used (which is double the number of available processors unless the <code>io.netty.eventLoopThreads</code> system property is set.	int	
<code>quarkus.sqs.async-client.event-loop.thread-name-prefix</code> The thread name prefix for threads created by this thread factory used by event loop group. The prefix will be appended with a number unique to the thread factory and a number unique to the thread. If not specified it defaults to <code>aws-java-sdk-NettyEventLoop</code>	string	



About the Duration format

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the [Duration#parse\(\) javadoc](#).

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.



About the MemorySize format

A size configuration option recognises string in this format (shown as a regular expression): `[0-9]+[KkMmGgTtPpEeZzYy]?`. If no suffix is given, assume bytes.