

Quarkus - Amazon S3 Client

Amazon S3 is an object storage service. It can be employed to store any type of object which allows for uses like storage for Internet applications, backup and recovery, disaster recovery, data archives, data lakes for analytics, any hybrid cloud storage. This extension provides functionality that allows the client to communicate with the service when running in Quarkus. You can find more information about S3 at [the Amazon S3 website](#).



The S3 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).

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 S3 locally and on AWS.

Prerequisites

To complete this guide, you need:

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- an IDE
- Apache Maven 3.5.3+
- [AWS Command line interface](#)
- An AWS Account to access the S3 service. Before you can use the AWS SDKs with Amazon S3, you must get an AWS access key ID and secret access key.
- Optionally, Docker for your system to run S3 locally for testing purposes

Provision S3 locally

The easiest way to start working with S3 is to run a local instance as a container.

```
docker run -it --publish 8008:4572 -e SERVICES=s3 -e START_WEB=0
localstack/localstack
```

This starts a S3 instance that is accessible on port `8008`.

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 S3 bucket

Create a S3 bucket using AWS CLI

```
aws s3 mb s3://quarkus.s3.quickstart --profile localstack
--endpoint-url=http://localhost:8008
```

Solution

The application built here allows to manage files stored in Amazon S3.

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-s3-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:1.7.0.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=amazon-s3-quickstart \
  -DclassName="org.acme.s3.S3SyncClientResource" \
  -Dpath="/s3" \
  -Dextensions="resteasy-jsonb,amazon-s3"
cd amazon-s3-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS and S3 Client extensions. After this, the `amazon-s3` extension has been added to your `pom.xml`.

Then, we'll add the following dependency to support `multipart/form-data` requests:

```
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-multipart-provider</artifactId>
</dependency>
```

Setting up the model

In this example, we will create an application to manage a list of files. The example application will demonstrate the two programming models supported by the extension.

Because the primary goal of our application is to upload a file into the S3 bucket, we need to setup the model we will be using to define the `multipart/form-data` payload, in the form of a `MultipartBody` POJO.

Create a `org.acme.s3.FormData` class as follows:

```
package org.acme.s3;

import java.io.InputStream;

import javax.ws.rs.FormParam;
import javax.ws.rs.core.MediaType;

import org.jboss.resteasy.annotations.providers.multipart.PartType;

public class FormData {

    @FormParam("file")
    @PartType(MediaType.APPLICATION_OCTET_STREAM)
    public InputStream data;

    @FormParam("filename")
    @PartType(MediaType.TEXT_PLAIN)
    public String fileName;

    @FormParam("mimetype")
    @PartType(MediaType.TEXT_PLAIN)
    public String mimeType;

}
```

The class defines three fields:

- `data` that fill capture stream of uploaded bytes from the client
- `fileName` that captures a filename as provided by the submitted form
- `mimeType` content type of the uploaded file

In the second step let's create a bean that will represent a file in a Amazon S3 bucket as follows:

```
package org.acme.s3;

import software.amazon.awssdk.services.s3.model.S3Object;

public class FileObject {
    private String objectKey;

    private Long size;

    public FileObject() {
    }

    public static FileObject from(S3Object s3Object) {
        FileObject file = new FileObject();
        if (s3Object != null) {
            file.setObjectKey(s3Object.key());
            file.setSize(s3Object.size());
        }
        return file;
    }

    public String getObjectKey() {
        return objectKey;
    }

    public Long getSize() {
        return size;
    }

    public FileObject setObjectKey(String objectKey) {
        this.objectKey = objectKey;
        return this;
    }

    public FileObject setSize(Long size) {
        this.size = size;
        return this;
    }
}
```

Nothing fancy. One important thing to note is that having a default constructor is required by the JSON serialization layer. The static `from` method creates a bean based on the `S3Object` object provided by the S3 client response when listing all the objects in a bucket.

Create JAX-RS resource

Now create a `org.acme.s3.CommonResource` that will consist of methods to prepare S3 request to get object from a S3 bucket, or to put file into a S3 bucket. Note a configuration property `bucket.name` is defined here as the request method required name of the S3 bucket.

```
package org.acme.s3;

import java.io.File;
import java.io.InputStream;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;
import java.util.Date;
import java.util.UUID;

import org.eclipse.microprofile.config.inject.ConfigProperty;

import software.amazon.awssdk.services.s3.model.GetObjectRequest;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;

abstract public class CommonResource {
    private final static String TEMP_DIR =
System.getProperty("java.io.tmpdir");

    @ConfigProperty(name = "bucket.name")
    String bucketName;

    protected PutObjectRequest buildPutRequest(FormData formData) {
        return PutObjectRequest.builder()
            .bucket(bucketName)
            .key(formData.fileName)
            .contentType(formData.mimeType)
            .build();
    }

    protected GetObjectRequest buildGetRequest(String objectKey) {
        return GetObjectRequest.builder()
            .bucket(bucketName)
            .key(objectKey)
            .build();
    }

    protected File tempFilePath() {
        return new File(TEMP_DIR, new
StringBuilder().append("s3AsyncDownloadedTemp")
            .append((new
Date()).getTime()).append(UUID.randomUUID())
            .append(".").append(".tmp").toString());
    }
}
```

```

    }

    protected File uploadToTemp(InputStream data) {
        File tempPath;
        try {
            tempPath = File.createTempFile("uploadS3Tmp", ".tmp");
            Files.copy(data, tempPath.toPath(),
StandardCopyOption.REPLACE_EXISTING);
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        }

        return tempPath;
    }
}

```

Then, create a `org.acme.s3.S3SyncClientResource` that will provides an API to upload/download files as well as to list all the files in a bucket.

```

package org.acme.s3;

import java.io.ByteArrayOutputStream;
import java.util.Comparator;
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.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;
import javax.ws.rs.core.Response.Status;
import javax.ws.rs.core.StreamingOutput;

import org.jboss.resteasy.annotations.jaxrs.PathParam;
import
org.jboss.resteasy.annotations.providers.multipart.MultipartForm;

import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.core.sync.ResponseTransformer;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.GetObjectResponse;
import software.amazon.awssdk.services.s3.model.ListObjectsRequest;
import software.amazon.awssdk.services.s3.model.PutObjectResponse;

```

```

import software.amazon.awssdk.services.s3.model.S3Object;

@Path("/s3")
public class S3SyncClientResource extends CommonResource {
    @Inject
    S3Client s3;

    @POST
    @Path("upload")
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    public Response uploadFile(@MultipartForm FormData formData)
    throws Exception {

        if (formData.fileName == null ||
        formData.fileName.isEmpty()) {
            return Response.status(Status.BAD_REQUEST).build();
        }

        if (formData.mimeType == null ||
        formData.mimeType.isEmpty()) {
            return Response.status(Status.BAD_REQUEST).build();
        }

        PutObjectResponse putResponse =
        s3.putObject(buildPutRequest(formData),
            RequestBody.fromFile(uploadToTemp(formData.data)));
        if (putResponse != null) {
            return Response.ok().status(Status.CREATED).build();
        } else {
            return Response.serverError().build();
        }
    }

    @GET
    @Path("download/{objectKey}")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    public Response downloadFile(@PathParam("objectKey") String
    objectKey) {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        GetObjectResponse object =
        s3.getObject(buildGetRequest(objectKey),
            ResponseTransformer.toOutputStream(baos));

        ResponseBuilder response = Response.ok((StreamingOutput)
        output -> baos.writeTo(output));
        response.header("Content-Disposition",
        "attachment;filename=" + objectKey);
        response.header("Content-Type", object.contentType());
        return response.build();
    }
}

```

```

    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<FileObject> listFiles() {
        ListObjectsRequest listRequest =
        ListObjectsRequest.builder().bucket(bucketName).build();

        //HEAD S3 objects to get metadata
        return
        s3.listObjects(listRequest).contents().stream().sorted(Comparator.c
        omparing(S3Object::lastModified).reversed())

        .map(FileObject::from).collect(Collectors.toList());
    }
}

```

Configuring S3 clients

Both S3 clients (sync and async) are configurable via the `application.properties` file that can be provided in the `src/main/resources` directory.



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 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.s3.sync-client.type=apache
```

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

```

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

```

For asynchronous client refer to [Going asynchronous](#) for more information.

If you're going to use a local S3 instance, configure it as follows:

```
quarkus.s3.endpoint-override=http://localhost:8008

quarkus.s3.aws.region=us-east-1
quarkus.s3.aws.credentials.type=static
quarkus.s3.aws.credentials.static-provider.access-key-id=test-key
quarkus.s3.aws.credentials.static-provider.secret-access-key=test-secret

bucket.name=quarkus.s3.quickstart
```

- `quarkus.s3.aws.region` - It's required by the client, but since you're using a local S3 instance you can pick any valid AWS region.
- `quarkus.s3.aws.credentials.type` - Set `static` credentials provider with any values for `access-key-id` and `secret-access-key`
- `quarkus.s3.endpoint-override` - Override the S3 client to use a local instance instead of an AWS service
- `bucket.name` - Name of the S3 bucket

If you want to work with an AWS account, you'd need to set it with:

```
bucket.name=<your-bucket-name>

quarkus.s3.aws.region=<YOUR_REGION>
quarkus.s3.aws.credentials.type=default
```

- `bucket.name` - name of the S3 bucket on your AWS account.
- `quarkus.s3.aws.region` you should set it to the region where your S3 bucket was created,
- `quarkus.s3.aws.credentials.type` - use the `default` credentials provider chain that looks for credentials in this order:
 - Java System Properties - `aws.accessKeyId` and `aws.secretAccessKey`
 - 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 ECS 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

Creating a frontend

Now let's add a simple web page to interact with our `S3SyncClientResource`. Quarkus automatically serves static resources located under the `META-INF/resources` directory. In the `src/main/resources/META-INF/resources` directory, add a `s3.html` file with the content from this [s3.html](#) file in it.

You can now interact with your REST service:

- start Quarkus with `./mvnw compile quarkus:dev`
- open a browser to <http://localhost:8080/s3.html>
- upload new file to the current S3 bucket via the form and see the list of files in the bucket

Next steps

Packaging

Packaging your application is as simple as `./mvnw clean package`. It can be run with `java -jar target/amazon-s3-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.s3.S3AsyncClientResource` that will be similar to our `S3SyncClientResource` but using an asynchronous programming model.

```
package org.acme.s3;

import java.io.File;
import java.util.Comparator;
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.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
```

```

import javax.ws.rs.core.Response.Status;

import org.jboss.resteasy.annotations.jaxrs.PathParam;
import
org.jboss.resteasy.annotations.providers.multipart.MultipartForm;

import io.smallrye.mutiny.Uni;
import software.amazon.awssdk.core.async.AsyncRequestBody;
import software.amazon.awssdk.core.async.AsyncResponseTransformer;
import software.amazon.awssdk.services.s3.S3AsyncClient;
import software.amazon.awssdk.services.s3.model.ListObjectsRequest;
import
software.amazon.awssdk.services.s3.model.ListObjectsResponse;
import software.amazon.awssdk.services.s3.model.S3Object;

@Path("/async-s3")
public class S3AsyncClientResource extends CommonResource {
    @Inject
    S3AsyncClient s3;

    @POST
    @Path("upload")
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    public Uni<Response> uploadFile(@MultipartForm FormData
formData) throws Exception {

        if (formData.fileName == null ||
formData.fileName.isEmpty()) {
            return
Uni.createFrom().item(Response.status(Status.BAD_REQUEST).build());
        }

        if (formData.mimeType == null ||
formData.mimeType.isEmpty()) {
            return
Uni.createFrom().item(Response.status(Status.BAD_REQUEST).build());
        }

        return Uni.createFrom()
            .completionStage(() -> {
                return s3.putObject(buildPutRequest(formData),
AsyncRequestBody.fromFile(uploadToTemp(formData.data)));
            })

        .onItem().ignore().andSwitchTo(Uni.createFrom().item(Response.creat
ed(null).build()))
            .onFailure().recoverWithItem(th -> {
                th.printStackTrace();
                return Response.serverError().build();
            })
    }
}

```

```

        });
    }

    @GET
    @Path("download/{objectKey}")
    @Produces(MediaType.APPLICATION_OCTET_STREAM)
    public Uni<Response> downloadFile(@PathParam("objectKey")
String objectKey) throws Exception {
        File tempFile = tempFilePath();

        return Uni.createFrom()
            .completionStage(() ->
s3.getObject(buildGetRequest(objectKey),
AsyncResponseTransformer.toFile(tempFile)))
            .onItem()
            .apply(object -> Response.ok(tempFile)
                .header("Content-Disposition",
"attachment;filename=" + objectKey)
                .header("Content-Type",
object.contentType()).build());
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Uni<List<FileObject>> listFiles() {
        ListObjectsRequest listRequest =
ListObjectsRequest.builder()
            .bucket(bucketName)
            .build();

        return Uni.createFrom().completionStage(() ->
s3.listObjects(listRequest))
            .onItem().transform(result -> toFileItems(result));
    }

    private List<FileObject> toFileItems(ListObjectsResponse
objects) {
        return objects.contents().stream()

            .sorted(Comparator.comparing(S3Object::lastModified).reversed())

            .map(FileObject::from).collect(Collectors.toList());
    }
}

```

You need the RESTEasy Mutiny support for asynchronous programming. Add the dependency to the `pom.xml`:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-resteasy-mutiny</artifactId>
</dependency>
```


Or you can alternatively run this command in your project base directory:



```
./mvnw quarkus:add-extension -Dextensions="resteasy-mutiny"
```

And 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.s3.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.s3.sync-client.type</code> Type of the sync HTTP client implementation | <code>url</code> , <code>apache</code> | <code>url</code> |
| <code>quarkus.s3.accelerate-mode</code> Enable using the accelerate endpoint when accessing S3. Accelerate endpoints allow faster transfer of objects by using Amazon CloudFront's globally distributed edge locations. | boolean | <code>false</code> |
| <code>quarkus.s3.checksum-validation</code> Enable doing a validation of the checksum of an object stored in S3. | boolean | <code>true</code> |

| | | |
|--|---------------|----------------|
| <code>quarkus.s3.chunked-encoding</code> | | |
| Enable using chunked encoding when signing the request payload for <code>software.amazon.awssdk.services.s3.model.PutObjectRequest</code> and <code>software.amazon.awssdk.services.s3.model.UploadPartRequest</code> . | boolean | true |
| <code>quarkus.s3.dualstack</code> | | |
| Enable dualstack mode for accessing S3. If you want to use IPv6 when accessing S3, dualstack must be enabled. | boolean | false |
| <code>quarkus.s3.path-style-access</code> | | |
| Enable using path style access for accessing S3 objects instead of DNS style access. DNS style access is preferred as it will result in better load balancing when accessing S3. | boolean | false |
| <code>quarkus.s3.use-arn-region-enabled</code> | | |
| Enable cross-region call to the region specified in the S3 resource ARN different than the region the client was configured with. If this flag is not set to 'true', the cross-region call will throw an exception. | boolean | false |
| <code>quarkus.s3.profile-name</code> | | |
| Define the profile name that should be consulted to determine the default value of <code>use-arn-region-enabled</code> . This is not used, if the <code>use-arn-region-enabled</code> is configured to 'true'. 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 | |
| AWS SDK client configurations | Type | Default |
| <code>quarkus.s3.endpoint-override</code> | | |
| The endpoint URI with which the SDK should communicate. If not specified, an appropriate endpoint to be used for the given service and region. | URI | |
| <code>quarkus.s3.api-call-timeout</code> | | |
| 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. | Duration ? | |

| | | |
|---|---|----------------|
| <code>quarkus.s3.api-call-attempt-timeout</code> 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. | Duration  | |
| AWS services configurations | Type | Default |
| <code>quarkus.s3.aws.region</code> An Amazon Web Services region that hosts the given service. It overrides region provider chain with static value of region with which the service client should communicate. If not set, region is retrieved via the default providers chain in the following order: <ul style="list-style-type: none"> • <code>aws.region</code> system property • <code>region</code> property from the profile file • Instance profile file See <code>software.amazon.awssdk.regions.Region</code> for available regions. | Region | |

`quarkus.s3.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.s3.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.s3.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.s3.aws.credentials.static-provider.access-key-id</code> AWS Access key id | string | |
| <code>quarkus.s3.aws.credentials.static-provider.secret-access-key</code> AWS Secret access key | string | |
| AWS Profile credentials provider configuration | Type | Default |
| <code>quarkus.s3.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.s3.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.s3.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.s3.aws.credentials.process-provider.process-output-limit</code> The maximum size of the output that can be returned by the external process before an exception is raised. | Memory Size  | 1024 |
| <code>quarkus.s3.aws.credentials.process-provider.command</code> The command that should be executed to retrieve credentials. | string | |
| Sync HTTP transport configurations | Type | Default |
| <code>quarkus.s3.sync-client.connection-timeout</code> The maximum amount of time to establish a connection before timing out. | Duration  | 2S |
| <code>quarkus.s3.sync-client.socket-timeout</code> The amount of time to wait for data to be transferred over an established, open connection before the connection is timed out. | Duration  | 30S |
| <code>quarkus.s3.sync-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.s3.sync-client.tls-managers-provider.file-store.path</code> Path to the key store. | path | |
| <code>quarkus.s3.sync-client.tls-managers-provider.file-store.type</code> Key store type. See the KeyStore section in the https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyStore[Java Cryptography Architecture Standard Algorithm Name Documentation] for information about standard keystore types. | string | |

| | | |
|--|---------------|----------------|
| <code>quarkus.s3.sync-client.tls-managers-provider.file-store.password</code> Key store password | string | |
| Apache HTTP client specific configurations | Type | Default |
| <code>quarkus.s3.sync-client.apache.connection-acquisition-timeout</code> The amount of time to wait when acquiring a connection from the pool before giving up and timing out. | Duration ? | 10S |
| <code>quarkus.s3.sync-client.apache.connection-max-idle-time</code> The maximum amount of time that a connection should be allowed to remain open while idle. | Duration ? | 60S |
| <code>quarkus.s3.sync-client.apache.connection-time-to-live</code> The maximum amount of time that a connection should be allowed to remain open, regardless of usage frequency. | Duration ? | |
| <code>quarkus.s3.sync-client.apache.max-connections</code> The maximum number of connections allowed in the connection pool. Each built HTTP client has its own private connection pool. | int | 50 |
| <code>quarkus.s3.sync-client.apache.expect-continue-enabled</code> Whether the client should send an HTTP expect-continue handshake before each request. | boolean | true |
| <code>quarkus.s3.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 | true |
| <code>quarkus.s3.sync-client.apache.proxy.enabled</code> Enable HTTP proxy | boolean | false |

| | | |
|--|----------------|----------------|
| <code>quarkus.s3.sync-client.apache.proxy.endpoint</code> | URI | |
| 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. | | |
| <code>quarkus.s3.sync-client.apache.proxy.username</code> | string | |
| The username to use when connecting through a proxy. | | |
| <code>quarkus.s3.sync-client.apache.proxy.password</code> | string | |
| The password to use when connecting through a proxy. | | |
| <code>quarkus.s3.sync-client.apache.proxy.ntlm-domain</code> | string | |
| For NTLM proxies - the Windows domain name to use when authenticating with the proxy. | | |
| <code>quarkus.s3.sync-client.apache.proxy.ntlm-workstation</code> | string | |
| For NTLM proxies - the Windows workstation name to use when authenticating with the proxy. | | |
| <code>quarkus.s3.sync-client.apache.proxy.preemptive-basic-authentication-enabled</code> | boolean | |
| Whether to attempt to authenticate preemptively against the proxy server using basic authentication. | | |
| <code>quarkus.s3.sync-client.apache.proxy.non-proxy-hosts</code> | list of string | |
| The hosts that the client is allowed to access without going through the proxy. | | |
| Netty HTTP transport configurations | Type | Default |
| <code>quarkus.s3.async-client.max-concurrency</code> | int | 50 |
| 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. | | |
| <code>quarkus.s3.async-client.max-pending-connection-acquires</code> | int | 10000 |
| The maximum number of pending acquires allowed. Once this exceeds, acquire tries will be failed. | | |

| | | |
|---|--|---------|
| <code>quarkus.s3.async-client.read-timeout</code> | Duration ? | 30S |
| The amount of time to wait for a read on a socket before an exception is thrown. Specify 0 to disable. | | |
| <code>quarkus.s3.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.s3.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.s3.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.s3.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.s3.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.s3.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.s3.async-client.protocol</code> | http1-1, http2 | http1-1 |
| The HTTP protocol to use. | | |
| <code>quarkus.s3.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.s3.async-client.http2.max-streams</code> The maximum number of concurrent streams for an HTTP/2 connection. This setting is only respected when the HTTP/2 protocol is used. | long | 4294967295 |
| <code>quarkus.s3.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.s3.async-client.http2.health-check-ping-period</code> Sets the period that the Netty client will send PING frames to the remote endpoint to check the health of the connection. To disable this feature, set a duration of 0. This setting is only respected when the HTTP/2 protocol is used. | Duration ? | 5 |
| <code>quarkus.s3.async-client.proxy.enabled</code> Enable HTTP proxy. | boolean | false |
| <code>quarkus.s3.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.s3.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.s3.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.s3.async-client.tls-managers-provider.file-store.path</code> Path to the key store. | path | |

| | | |
|--|---------|--------------------|
| <code>quarkus.s3.async-client.tls-managers-provider.file-store.type</code> | | |
| Key store type. See the KeyStore section in the https://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyStore[Java Cryptography Architecture Standard Algorithm Name Documentation] for information about standard keystore types. | string | |
| <code>quarkus.s3.async-client.tls-managers-provider.file-store.password</code> | | |
| Key store password | string | |
| <code>quarkus.s3.async-client.event-loop.override</code> | | |
| Enable the custom configuration of the Netty event loop group. | boolean | <code>false</code> |
| <code>quarkus.s3.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.s3.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.