Quarkus - Amazon DynamoDB Client

DynamoDB is a scalable AWS managed **NoSQL** database. It supports both key-value and document data models, that enables to have a flexible schema for your data. This extension provides functionality that allows the client to communicate with the service when running in Quarkus. You can find more information about DynamoDB at the Amazon DynamoDB website.

> ℹ️ The DynamoDB 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). Keep in mind it's actively developed and does not support yet all the features available in SDK 1.x such as Document APIs or Object Mappers

> ℹ️ 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 the DynamoDB 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 DynamoDB service

- Optionally, Docker for your system to run DynamoDB locally for testing purposes

Setup DynamoDB locally

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

```
docker run --publish 8000:8000 amazon/dynamodb-local:1.11.477 -jar
DynamoDBLocal.jar -inMemory -sharedDb
```

This starts a DynamoDB instance that is accessible on port 8000. You can check it's running by
accessing the web shell on http://localhost:8000/shell.

Have a look at the Setting Up DynamoDB Local guide for other options to run DynamoDB.

Open http://localhost:8000/shell in your browser.

Copy and paste the following code to the shell and run it:

```
var params = {
    TableName: 'QuarkusFruits',
    KeySchema: [{ AttributeName: 'fruitName', KeyType: 'HASH' }],
    AttributeDefinitions: [{  AttributeName: 'fruitName',
AttributeType: 'S', }],
    ProvisionedThroughput: { ReadCapacityUnits: 1,
WriteCapacityUnits: 1, }
};

dynamodb.createTable(params, function(err, data) {
    if (err) ppJson(err);
    else ppJson(data);

});
```

Set up Dynamodb on AWS

Before you can use the AWS SDKs with DynamoDB, you must get an AWS access key ID and secret
access key. For more information, see Setting Up DynamoDB (Web Service).

We recommend to use the AWS CLI to provision the table:

```
aws dynamodb create-table --table-name QuarkusFruits \
                        --attribute-definitions
AttributeName=fruitName,AttributeType=S \
                        --key-schema
AttributeName=fruitName,KeyType=HASH \
                        --provisioned-throughput
ReadCapacityUnits=1,WriteCapacityUnits=1
```

Solution

The application built here allows to manage elements (fruits) stored in Amazon DynamoDB.

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 `dynamodb-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:1.3.1.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=dynamodb-quickstart \
    -DclassName="org.acme.dynamodb.FruitResource" \
    -Dpath="/fruits" \
    -Dextensions="resteasy-jsonb,dynamodb,resteasy-mutiny"
cd dynamodb-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS, Mutiny and DynamoDB Client extensions. After this, the `amazon-dynamodb` 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 to manage a list of fruits. The example application will demonstrate the two programming models supported by the extension.

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

```
package org.acme.dynamodb;

import java.util.Map;
import java.util.Objects;

import io.quarkus.runtime.annotations.RegisterForReflection;
import software.amazon.awssdk.services.dynamodb.model.AttributeValue;

@RegisterForReflection
public class Fruit {
```

```java
    private String name;
    private String description;

    public Fruit() {
    }

    public static Fruit from(Map<String, AttributeValue> item) {
        Fruit fruit = new Fruit();
        if (item != null && !item.isEmpty()) {
            fruit.setName(item.get(AbstractService.FRUIT_NAME_COL)
.s());
            fruit.setDescription(item.get(AbstractService
.FRUIT_DESC_COL).s());
        }
        return fruit;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

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

        Fruit other = (Fruit) obj;

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

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

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 `Map` object provided by the DynamoDB client response.

Now create a `org.acme.dynamodb.AbstractService` that will consist of helper methods that prepare DynamoDB request objects for reading and adding items to the table.

```java
package org.acme.dynamodb;

import java.util.HashMap;
import java.util.Map;

import software.amazon.awssdk.services.dynamodb.model.AttributeValue;
import software.amazon.awssdk.services.dynamodb.model.GetItemRequest;
import software.amazon.awssdk.services.dynamodb.model.PutItemRequest;
import software.amazon.awssdk.services.dynamodb.model.ScanRequest;

public abstract class AbstractService {

    public final static String FRUIT_NAME_COL = "fruitName";
    public final static String FRUIT_DESC_COL = "fruitDescription";

    public String getTableName() {
        return "QuarkusFruits";
    }

    protected ScanRequest scanRequest() {
        return ScanRequest.builder().tableName(getTableName())
                .attributesToGet(FRUIT_NAME_COL, FRUIT_DESC_COL).build();
    }

    protected PutItemRequest putRequest(Fruit fruit) {
        Map<String, AttributeValue> item = new HashMap<>();
        item.put(FRUIT_NAME_COL, AttributeValue.builder().s(fruit.getName()).build());
        item.put(FRUIT_DESC_COL, AttributeValue.builder().s(fruit.getDescription()).build());

        return PutItemRequest.builder()
                .tableName(getTableName())
                .item(item)
                .build();
    }

    protected GetItemRequest getRequest(String name) {
```

```
        Map<String, AttributeValue> key = new HashMap<>();
        key.put(FRUIT_NAME_COL, AttributeValue.builder().s(name)
.build());

        return GetItemRequest.builder()
                .tableName(getTableName())
                .key(key)
                .attributesToGet(FRUIT_NAME_COL, FRUIT_DESC_COL)
                .build();
    }
}
```

Then, create a `org.acme.dynamodb.FruitSyncService` that will be the business layer of our application and stores/loads the fruits from DynamoDB using the synchronous client.

```
package org.acme.dynamodb;

import java.util.List;
import java.util.stream.Collectors;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import software.amazon.awssdk.services.dynamodb.DynamoDbClient;

@ApplicationScoped
public class FruitSyncService extends AbstractService {

    @Inject
    DynamoDbClient dynamoDB;

    public List<Fruit> findAll() {
        return dynamoDB.scanPaginator(scanRequest()).items().
stream()
                .map(Fruit::from)
                .collect(Collectors.toList());
    }

    public List<Fruit> add(Fruit fruit) {
        dynamoDB.putItem(putRequest(fruit));
        return findAll();
    }

    public Fruit get(String name) {
        return Fruit.from(dynamoDB.getItem(getRequest(name)).item(
));
    }
}
```

Now, edit the `org.acme.dynamodb.FruitResource` class as follows:

```java
package org.acme.dynamodb;

import java.util.List;

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.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitResource {

    @Inject
    FruitSyncService service;

    @GET
    public List<Fruit> getAll() {
        return service.findAll();
    }

    @GET
    @Path("{name}")
    public Fruit getSingle(@PathParam("name") String name) {
        return service.get(name);
    }

    @POST
    public List<Fruit> add(Fruit fruit) {
        service.add(fruit);
        return getAll();
    }
}
```

The implementation is pretty straightforward and you just need to define your endpoints using the JAX-RS annotations and use the `FruitSyncService` to list/add new fruits.

Configuring DynamoDB clients

Both DynamoDB 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 added to the

classpath a proper implementation of the sync client. By default the extension uses URL connection HTTP client, so you need to add a URL connection client dependency to the `pom.xml` file:

```xml
<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.dynamodb.sync-client.type=apache
```

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

```xml
<dependency>
    <groupId>software.amazon.awssdk</groupId>
    <artifactId>apache-client</artifactId>
    <exclusions>
        <exclusion>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

> ℹ️ It's important to exclude `commons-logging` from the client dependency to force Apache HTTP client to use Quarkus logger.

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

```
quarkus.dynamodb.endpoint-override=http://localhost:8000

quarkus.dynamodb.aws.region=eu-central-1
quarkus.dynamodb.aws.credentials.type=static
quarkus.dynamodb.aws.credentials.static-provider.access-key-id=test-key
quarkus.dynamodb.aws.credentials.static-provider.secret-access-key=test-secret
```

- `quarkus.dynamodb.aws.region` - It's required by the client, but since you're using a local DynamoDB instance you can pick any valid AWS region.

- `quarkus.dynamodb.aws.credentials.type` - Set `static` credentials provider with any values for `access-key-id` and `secret-access-key`

- `quarkus.dynamodb.endpoint-override` - Override the DynamoDB client to use a local instance instead of an AWS service

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

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

- `quarkus.dynamodb.aws.region` you should set it to the region where you provisioned the DynamoDB table,
- `quarkus.dynamodb.aws.credentials.type` - 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

Next steps

Packaging

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

```java
package org.acme.dynamodb;

import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.stream.Collectors;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import io.smallrye.mutiny.Uni;
import software.amazon.awssdk.services.dynamodb.DynamoDbAsyncClient;

@ApplicationScoped
public class FruitAsyncService extends AbstractService {

    @Inject
    DynamoDbAsyncClient dynamoDB;

    public Uni<List<Fruit>> findAll() {
        return Uni.createFrom().completionStage(() -> dynamoDB.scan(scanRequest()))
                .onItem().apply(res -> res.items().stream().map(Fruit::from).collect(Collectors.toList()));
    }

    public Uni<List<Fruit>> add(Fruit fruit) {
        return Uni.createFrom().completionStage(() -> dynamoDB.putItem(putRequest(fruit)))
                .onItem().ignore().andSwitchTo(this::findAll);
    }

    public Uni<Fruit> get(String name) {
        return Uni.createFrom().completionStage(() -> dynamoDB.getItem(getRequest(name)))
                .onItem().apply(resp -> Fruit.from(resp.item()));
    }
}
```

In the previous code, we create `Uni` instances from the `CompletionStage` objects returned by the asynchronous DynamoDB client, and then transform the emitted item.

Then, create an asynchronous REST resource that consumes this async service:

```java
package org.acme.dynamodb;

import io.smallrye.mutiny.Uni;

import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import java.util.List;

@Path("/async-fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitAsyncResource {

    @Inject
    FruitAsyncService service;

    @GET
    public Uni<List<Fruit>> getAll() {
        return service.findAll();
    }

    @GET
    @Path("{name}")
    public Uni<Fruit> getSingle(@PathParam("name") String name) {
        return service.get(name);
    }

    @POST
    public Uni<List<Fruit>> add(Fruit fruit) {
        return service.add(fruit)
                .onItem().ignore().andSwitchTo(this::getAll);
    }
}
```

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

```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
🔒 `quarkus.dynamodb.interceptors` 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 `software.amazon.awssdk.core.interceptor.ExecutionIntercept or` interface.	list of class name	
🔒 `quarkus.dynamodb.sync-client.type` Type of the sync HTTP client implementation	`url`, `apache`	`url`
`quarkus.dynamodb.enable-endpoint-discovery` Enable DynamoDB service endpoint discovery.	boolean	`false`
`quarkus.dynamodb.endpoint-override` The endpoint URI with which the SDK should communicate. If not specified, an appropriate endpoint to be used for DynamoDB service and region.	URI	
`quarkus.dynamodb.api-call-timeout` 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 ❓	
`quarkus.dynamodb.api-call-attempt-timeout` 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 ❓	

`quarkus.dynamodb.aws.region` An Amazon Web Services region that hosts DynamoDB. It overrides region provider chain with static value of region with which the DynamoDB client should communicate. If not set, region is retrieved via the default providers chain in the following order: • `aws.region` system property • `region` property from the profile file • Instance profile file See `software.amazon.awssdk.regions.Region` for available regions.	Region	

`quarkus.dynamodb.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-property, env-variable, profile, container, instance-profile, process, anonymous`	`default`

`quarkus.dynamodb.aws.credentials.default-provider.async-credential-update-enabled` Whether this provider should fetch credentials asynchronously in the background. If this is `true`, threads are less likely to block, but additional resources are used to maintain the provider.	boolean	`false`
`quarkus.dynamodb.aws.credentials.default-provider.reuse-last-provider-enabled` 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	`true`
`quarkus.dynamodb.aws.credentials.static-provider.access-key-id` AWS Access key id	string	
`quarkus.dynamodb.aws.credentials.static-provider.secret-access-key` AWS Secret access key	string	
`quarkus.dynamodb.aws.credentials.profile-provider.profile-name` The name of the profile that should be used by this credentials provider. If not specified, the value in `AWS_PROFILE` environment variable or `aws.profile` system property is used and defaults to `default` name.	string	
`quarkus.dynamodb.aws.credentials.process-provider.async-credential-update-enabled` 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	`false`
`quarkus.dynamodb.aws.credentials.process-provider.credential-refresh-threshold` 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 ❓	`15S`

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

`quarkus.dynamodb.sync-client.apache.use-idle-connection-reaper` Whether the idle connections in the connection pool should be closed asynchronously. When enabled, connections left idling for longer than `quarkus.dynamodb.sync-client.connection-max-idle-time` will be closed. This will not close connections currently in use.	boolean	true
`quarkus.dynamodb.sync-client.apache.proxy.enabled` Enable HTTP proxy	boolean	false
`quarkus.dynamodb.sync-client.apache.proxy.endpoint` 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	
`quarkus.dynamodb.sync-client.apache.proxy.username` The username to use when connecting through a proxy.	string	
`quarkus.dynamodb.sync-client.apache.proxy.password` The password to use when connecting through a proxy.	string	
`quarkus.dynamodb.sync-client.apache.proxy.ntlm-domain` For NTLM proxies - the Windows domain name to use when authenticating with the proxy.	string	
`quarkus.dynamodb.sync-client.apache.proxy.ntlm-workstation` For NTLM proxies - the Windows workstation name to use when authenticating with the proxy.	string	
`quarkus.dynamodb.sync-client.apache.proxy.preemptive-basic-authentication-enabled` Whether to attempt to authenticate preemptively against the proxy server using basic authentication.	boolean	
`quarkus.dynamodb.sync-client.apache.proxy.non-proxy-hosts` The hosts that the client is allowed to access without going through the proxy.	list of string	

`quarkus.dynamodb.sync-client.apache.tls-managers-provider.type` TLS managers provider type. Available providers: • `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 `javax.net.ssl.keyStore`, `javax.net.ssl.keyStorePassword`, and `javax.net.ssl.keyStoreType` properties defined by the JSSE. • `file-store` - Provider that loads a the key store from a file.	`none`, `system-property`, `file-store`	`system-property`
`quarkus.dynamodb.async-client.max-concurrency` 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`
`quarkus.dynamodb.async-client.max-pending-connection-acquires` The maximum number of pending acquires allowed. Once this exceeds, acquire tries will be failed.	int	`10000`
`quarkus.dynamodb.async-client.read-timeout` The amount of time to wait for a read on a socket before an exception is thrown. Specify `0` to disable.	Duration ❓	`30S`
`quarkus.dynamodb.async-client.write-timeout` The amount of time to wait for a write on a socket before an exception is thrown. Specify `0` to disable.	Duration ❓	`30S`
`quarkus.dynamodb.async-client.connection-timeout` The amount of time to wait when initially establishing a connection before giving up and timing out.	Duration ❓	`10S`
`quarkus.dynamodb.async-client.connection-acquisition-timeout` The amount of time to wait when acquiring a connection from the pool before giving up and timing out.	Duration ❓	`2S`

`quarkus.dynamodb.async-client.connection-time-to-live` The maximum amount of time that a connection should be allowed to remain open, regardless of usage frequency.	Duration ❓	
`quarkus.dynamodb.async-client.connection-max-idle-time` The maximum amount of time that a connection should be allowed to remain open while idle. Currently has no effect if `quarkus.dynamodb.async-client.use-idle-connection-reaper` is false.	Duration ❓	`60S`
`quarkus.dynamodb.async-client.use-idle-connection-reaper` Whether the idle connections in the connection pool should be closed. When enabled, connections left idling for longer than `quarkus.dynamodb.async-client.connection-max-idle-time` will be closed. This will not close connections currently in use.	boolean	`true`
`quarkus.dynamodb.async-client.protocol` The HTTP protocol to use.	`http1-1`, `http2`	`http1-1`
`quarkus.dynamodb.async-client.max-http2-streams` The maximum number of concurrent streams for an HTTP/2 connection. This setting is only respected when the HTTP/2 protocol is used. 0 means unlimited.	int	`0`
`quarkus.dynamodb.async-client.ssl-provider` The SSL Provider to be used in the Netty client. Default is `OPENSSL` if available, `JDK` otherwise.	`jdk`, `openssl`, `openssl-refcnt`	
`quarkus.dynamodb.async-client.proxy.enabled` Enable HTTP proxy.	boolean	`false`
`quarkus.dynamodb.async-client.proxy.endpoint` 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	
`quarkus.dynamodb.async-client.proxy.non-proxy-hosts` The hosts that the client is allowed to access without going through the proxy.	list of string	

`quarkus.dynamodb.async-client.tls-managers-provider.type` TLS managers provider type. Available providers: • `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 `javax.net.ssl.keyStore`, `javax.net.ssl.keyStorePassword`, and `javax.net.ssl.keyStoreType` properties defined by the JSSE. • `file-store` - Provider that loads a the key store from a file.	`none, system-property, file-store`	`system-property`
`quarkus.dynamodb.async-client.event-loop.override` Enable the custom configuration of the Netty event loop group.	boolean	`false`
`quarkus.dynamodb.async-client.event-loop.number-of-threads` 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 `io.netty.eventLoopThreads` system property is set.	int	
`quarkus.dynamodb.async-client.event-loop.thread-name-prefix` 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 `aws-java-sdk-NettyEventLoop`	string	
Configuration of the file store provider This configuration section is optional	**Type**	**Default**
`quarkus.dynamodb.sync-client.apache.tls-managers-provider.file-store.path` Path to the key store.	path	required ⊘
`quarkus.dynamodb.sync-client.apache.tls-managers-provider.file-store.type` Key store type. See the KeyStore section in the Java Cryptography Architecture Standard Algorithm Name Documentation for information about standard keystore types.	string	required ⊘

`quarkus.dynamodb.sync-client.apache.tls-managers-provider.file-store.password` Key store password	string	required ⊘
`quarkus.dynamodb.async-client.tls-managers-provider.file-store.path` Path to the key store.	path	required ⊘
`quarkus.dynamodb.async-client.tls-managers-provider.file-store.type` Key store type. See the KeyStore section in the Java Cryptography Architecture Standard Algorithm Name Documentation for information about standard keystore types.	string	required ⊘
`quarkus.dynamodb.async-client.tls-managers-provider.file-store.password` Key store password	string	required ⊘

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.