Quarkus - Application Data Caching

In this guide, you will learn how to enable application data caching in any CDI managed bean of your Quarkus application.

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.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3

Scenario

Let's imagine you want to expose in your Quarkus application a REST API that allows users to retrieve the weather forecast for the next three days. The problem is that you have to rely on an external meteorological service which only accepts requests for one day at a time and takes forever to answer. Since the weather forecast is updated once every twelve hours, caching the service responses would definitely improve your API performances.

We'll do that using a single Quarkus annotation.

Solution

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 `cache-quickstart` directory.

Creating the Maven project

First, we need to create a new Quarkus project using Maven with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.7.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=cache-quickstart \
    -DclassName="org.acme.cache.WeatherForecastResource" \
    -Dpath="/weather" \
    -Dextensions="cache,resteasy-jsonb"
```

This command generates the Maven project with a REST endpoint and imports the `cache` and `resteasy-jsonb` extensions.

If you already have your Quarkus project configured, you can add the `cache` extension to your project by running the following command in your project base directory:

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

This will add the following to your `pom.xml`:

```xml
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-cache</artifactId>
</dependency>
```

Creating the REST API

Let's start by creating a service that will simulate an extremely slow call to the external meteorological service. Create `src/main/java/org/acme/cache/WeatherForecastService.java` with the following content:

```java
package org.acme.cache;

import java.time.LocalDate;

import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class WeatherForecastService {

    public String getDailyForecast(LocalDate date, String city) {
        try {
            Thread.sleep(2000L); ①
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return date.getDayOfWeek() + " will be " +
getDailyResult(date.getDayOfMonth() % 4) + " in " + city;
    }

    private String getDailyResult(int dayOfMonthModuloFour) {
        switch (dayOfMonthModuloFour) {
            case 0:
                return "sunny";
            case 1:
                return "cloudy";
            case 2:
                return "chilly";
            case 3:
                return "rainy";
            default:
                throw new IllegalArgumentException();
        }
    }
}
```

① This is where the slowness comes from.

We also need a class that will contain the response sent to the users when they ask for the next three days weather forecast. Create `src/main/java/org/acme/cache/WeatherForecast.java` this way:

```java
package org.acme.cache;

import java.util.List;

public class WeatherForecast {

    private List<String> dailyForecasts;

    private long executionTimeInMs;

    public WeatherForecast(List<String> dailyForecasts, long
executionTimeInMs) {
        this.dailyForecasts = dailyForecasts;
        this.executionTimeInMs = executionTimeInMs;
    }

    public List<String> getDailyForecasts() {
        return dailyForecasts;
    }

    public long getExecutionTimeInMs() {
        return executionTimeInMs;
    }
}
```

Now, we just need to update the generated `WeatherForecastResource` class to use the service and response:

```java
package org.acme.cache;

import java.time.LocalDate;
import java.util.Arrays;
import java.util.List;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.jboss.resteasy.annotations.jaxrs.QueryParam;

@Path("/weather")
public class WeatherForecastResource {

    @Inject
    WeatherForecastService service;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public WeatherForecast getForecast(@QueryParam String city,
@QueryParam long daysInFuture) { ①
        long executionStart = System.currentTimeMillis();
        List<String> dailyForecasts = Arrays.asList(

service.getDailyForecast(LocalDate.now().plusDays(daysInFuture),
city),

service.getDailyForecast(LocalDate.now().plusDays(daysInFuture +
1L), city),

service.getDailyForecast(LocalDate.now().plusDays(daysInFuture +
2L), city)
        );
        long executionEnd = System.currentTimeMillis();
        return new WeatherForecast(dailyForecasts, executionEnd -
executionStart);
    }
}
```

① If the `daysInFuture` query parameter is omitted, the three days weather forecast will start from the current day. Otherwise, it will start from the current day plus the `daysInFuture` value.

We're all done! Let's check if everything's working.

First, run the application using `./mvnw compile quarkus:dev` from the project directory.

Then, call `http://localhost:8080/weather?city=Raleigh` from a browser. After six long seconds, the application will answer something like this:

```
{"dailyForecasts":["MONDAY will be cloudy in Raleigh","TUESDAY will
be chilly in Raleigh","WEDNESDAY will be rainy in
Raleigh"],"executionTimeInMs":6001}
```

💡 | The response content may vary depending on the day you run the code.

You can try calling the same URL again and again, it will always take six seconds to answer.

Enabling the cache

Now that your Quarkus application is up and running, let's tremendously improve its response time by caching the external meteorological service responses. Update the `WeatherForecastService` class like this:

```
package org.acme.cache;

import java.time.LocalDate;

import javax.enterprise.context.ApplicationScoped;

import io.quarkus.cache.CacheResult;

@ApplicationScoped
public class WeatherForecastService {

    @CacheResult(cacheName = "weather-cache") ①
    public String getDailyForecast(LocalDate date, String city) {
        try {
            Thread.sleep(2000L);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        return date.getDayOfWeek() + " will be " +
getDailyResult(date.getDayOfMonth() % 4) + " in " + city;
    }

    private String getDailyResult(int dayOfMonthModuloFour) {
        switch (dayOfMonthModuloFour) {
            case 0:
                return "sunny";
            case 1:
                return "cloudy";
            case 2:
                return "chilly";
            case 3:
                return "rainy";
            default:
                throw new IllegalArgumentException();
        }
    }
}
```

① We only added this annotation (and the associated import of course).

Let's try to call `http://localhost:8080/weather?city=Raleigh` again. You're still waiting a long time before receiving an answer. This is normal since the server just restarted and the cache was empty.

Wait a second! The server restarted by itself after the `WeatherForecastService` update? Yes, this is one of Quarkus amazing features for developers called `live coding`.

Now that the cache was loaded during the previous call, try calling the same URL. This time, you should get a super fast answer with an `executionTimeInMs` value close to 0.

Let's see what happens if we start from one day in the future using the `http://localhost:8080/weather?city=Raleigh&daysInFuture=1` URL. You should get an answer two seconds later since two of the requested days were already loaded in the cache.

You can also try calling the same URL with a different city and see the cache in action again. The first call will take six seconds and the following ones will be answered immediately.

Congratulations! You just added application data caching to your Quarkus application with a single line of code!

Do you want to learn more about the Quarkus application data caching abilities? The following sections will show you everything there is to know about it.

Caching annotations

Quarkus offers a set of annotations that can be used in a CDI managed bean to enable caching abilities.

@CacheResult

Loads a method result from the cache without executing the method body whenever possible.

When a method annotated with `@CacheResult` is invoked, Quarkus will compute a cache key and use it to check in the cache whether the method has been already invoked. If the method has one or more arguments, the key computation is done from all the method arguments if none of them is annotated with `@CacheKey`, or all the arguments annotated with `@CacheKey` otherwise. Each non-primitive method argument that is part of the key must implement `equals()` and `hashCode()` correctly for the cache to work as expected. This annotation can also be used on a method with no arguments, a default key derived from the cache name is generated in that case. If a value is found in the cache, it is returned and the annotated method is never actually executed. If no value is found, the annotated method is invoked and the returned value is stored in the cache using the computed or generated key.

A method annotated with `CacheResult` is protected by a lock on cache miss mechanism. If several concurrent invocations try to retrieve a cache value from the same missing key, the method will only be invoked once. The first concurrent invocation will trigger the method invocation while the subsequent concurrent invocations will wait for the end of the method invocation to get the cached result. The `lockTimeout` parameter can be used to interrupt the lock after a given delay. The lock timeout is disabled by default, meaning the lock is never interrupted. See the parameter Javadoc for more details.

This annotation cannot be used on a method returning `void`.

> Quarkus is able to also cache `null` values unlike the underlying Caffeine provider. See more on this topic below.

@CacheInvalidate

Removes an entry from the cache.

When a method annotated with @CacheInvalidate is invoked, Quarkus will compute a cache key and use it to try to remove an existing entry from the cache. If the method has one or more arguments, the key computation is done from all the method arguments if none of them is annotated with @CacheKey, or all the arguments annotated with @CacheKey otherwise. This annotation can also be used on a method with no arguments, a default key derived from the cache name is generated in that case. If the key does not identify any cache entry, nothing will happen.

> If the @CacheResult or @CacheInvalidate annotations are used on a method with no parameters, a unique default cache key derived from the cache name will be generated and used.

@CacheInvalidateAll

When a method annotated with @CacheInvalidateAll is invoked, Quarkus will remove all entries from the cache.

@CacheKey

When a method argument is annotated with @CacheKey, it is identified as a part of the cache key during an invocation of a method annotated with @CacheResult or @CacheInvalidate.

This annotation is optional and should only be used when some of the method arguments are NOT part of the cache key.

Composite cache key building logic

When a cache key is built from several method arguments, whether they are explicitly identified with @CacheKey or not, the building logic depends on the order of these arguments in the method signature. On the other hand, the arguments names are not used at all and do not have any effect on the cache key.

```
package org.acme.cache;

import javax.enterprise.context.ApplicationScoped;

import io.quarkus.cache.CacheInvalidate;
import io.quarkus.cache.CacheResult;

@ApplicationScoped
public class CachedService {

    @CacheResult(cacheName = "foo")
    public Object load(String keyElement1, Integer keyElement2) {
        // Call expensive service here.
    }

    @CacheInvalidate(cacheName = "foo")
    public void invalidate1(String keyElement2, Integer
keyElement1) { ①
    }

    @CacheInvalidate(cacheName = "foo")
    public void invalidate2(Integer keyElement2, String
keyElement1) { ②
    }

    @CacheInvalidate(cacheName = "foo")
    public void invalidate3(Object notPartOfTheKey, @CacheKey
String keyElement1, @CacheKey Integer keyElement2) { ③
    }

    @CacheInvalidate(cacheName = "foo")
    public void invalidate4(Object notPartOfTheKey, @CacheKey
Integer keyElement2, @CacheKey String keyElement1) { ④
    }
}
```

① Calling this method WILL invalidate values cached by the `load` method even if the key elements names have been swapped.

② Calling this method WILL NOT invalidate values cached by the `load` method because the key elements order is different.

③ Calling this method WILL invalidate values cached by the `load` method because the key elements order is the same.

④ Calling this method WILL NOT invalidate values cached by the `load` method because the key elements order is different.

Configuring the underlying caching provider

This extension uses Caffeine as its underlying caching provider. Caffeine is a high performance, near optimal caching library.

Caffeine configuration properties

Each of the Caffeine caches backing up the Quarkus application data caching extension can be configured using the following properties in the `application.properties` file. By default caches do not perform any type of eviction if not configured.

> 💡 You need to replace `cache-name` in all of the following properties with the real name of the cache you want to configure.

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

Configuration property	Type	Default
🔒 `quarkus.cache.caffeine."cache-name".initial-capacity` Minimum total size for the internal data structures. Providing a large enough estimate at construction time avoids the need for expensive resizing operations later, but setting this value unnecessarily high wastes memory.	int	
🔒 `quarkus.cache.caffeine."cache-name".maximum-size` Maximum number of entries the cache may contain. Note that the cache **may evict an entry before this limit is exceeded or temporarily exceed the threshold while evicting**. As the cache size grows close to the maximum, the cache evicts entries that are less likely to be used again. For example, the cache may evict an entry because it hasn't been used recently or very often.	long	
🔒 `quarkus.cache.caffeine."cache-name".expire-after-write` Specifies that each entry should be automatically removed from the cache once a fixed duration has elapsed after the entry's creation, or the most recent replacement of its value.	Duration ❓	
🔒 `quarkus.cache.caffeine."cache-name".expire-after-access` Specifies that each entry should be automatically removed from the cache once a fixed duration has elapsed after the entry's creation, the most recent replacement of its value, or its last read.	Duration ❓	

Here's what your cache configuration could look like:

```
quarkus.cache.caffeine."foo".initial-capacity=10 ①
quarkus.cache.caffeine."foo".maximum-size=20
quarkus.cache.caffeine."foo".expire-after-write=60S
quarkus.cache.caffeine."bar".maximum-size=1000 ②
```

① The `foo` cache is being configured.

② The `bar` cache is being configured.

Context propagation

This extension relies on non-blocking calls internally for cache values computations. By default, there's no context propagation between the calling thread (from your application) and a thread that performs such a computation.

The context propagation can be enabled for this extension by simply adding the `quarkus-smallrye-context-propagation` extension to your project.

If you see a `javax.enterprise.context.ContextNotActiveException` in your application log during a cache computation, then you probably need to enable the context propagation.

You can find more information about context propagation in Quarkus in the dedicated guide.

Annotated beans examples

Implicit simple cache key

```
package org.acme.cache;

import javax.enterprise.context.ApplicationScoped;

import io.quarkus.cache.CacheInvalidate;
import io.quarkus.cache.CacheInvalidateAll;
import io.quarkus.cache.CacheResult;

@ApplicationScoped
public class CachedService {

    @CacheResult(cacheName = "foo")
    public Object load(Object key) { ①
        // Call expensive service here.
    }

    @CacheInvalidate(cacheName = "foo")
    public void invalidate(Object key) { ①
    }

    @CacheInvalidateAll(cacheName = "foo")
    public void invalidateAll() {
    }
}
```

① The cache key is implicit since there's no @CacheKey annotation.

Explicit composite cache key

```
package org.acme.cache;

import javax.enterprise.context.Dependent;

import io.quarkus.cache.CacheInvalidate;
import io.quarkus.cache.CacheInvalidateAll;
import io.quarkus.cache.CacheKey;
import io.quarkus.cache.CacheResult;

@Dependent
public class CachedService {

    @CacheResult(cacheName = "foo")
    public String load(@CacheKey Object keyElement1, @CacheKey
Object keyElement2, Object notPartOfTheKey) { ①
        // Call expensive service here.
    }

    @CacheInvalidate(cacheName = "foo")
    public void invalidate(@CacheKey Object keyElement1, @CacheKey
Object keyElement2, Object notPartOfTheKey) { ①
    }

    @CacheInvalidateAll(cacheName = "foo")
    public void invalidateAll() {
    }
}
```

① The cache key is explicitly composed of two elements. The method signature also contains a third argument which is not part of the key.

Default cache key

```
package org.acme.cache;

import javax.enterprise.context.Dependent;

import io.quarkus.cache.CacheInvalidate;
import io.quarkus.cache.CacheInvalidateAll;
import io.quarkus.cache.CacheResult;

@Dependent
public class CachedService {

    @CacheResult(cacheName = "foo")
    public String load() { ①
        // Call expensive service here.
    }

    @CacheInvalidate(cacheName = "foo")
    public void invalidate() { ①
    }

    @CacheInvalidateAll(cacheName = "foo")
    public void invalidateAll() {
    }
}
```

① A unique default cache key derived from the cache name is generated and used.

Multiple annotations on a single method

```
package org.acme.cache;

import javax.inject.Singleton;

import io.quarkus.cache.CacheInvalidate;
import io.quarkus.cache.CacheInvalidateAll;
import io.quarkus.cache.CacheResult;

@Singleton
public class CachedService {

    @CacheInvalidate(cacheName = "foo")
    @CacheResult(cacheName = "foo")
    public String forceCacheEntryRefresh(Object key) { ①
        // Call expensive service here.
    }

    @CacheInvalidateAll(cacheName = "foo")
    @CacheInvalidateAll(cacheName = "bar")
    public void multipleInvalidateAll(Object key) { ②
    }
}
```

① This method can be used to force a refresh of the cache entry corresponding to the given key.

② This method will invalidate all entries from the `foo` and `bar` caches with a single call.

Negative caching and nulls

Sometimes one wants to cache the results of an (expensive) remote call. If the remote call fails, one may not want to cache the result or exception, but rather re-try the remote call on the next invocation.

A simple approach could be to catch the exception and return `null`, so that the caller can act accordingly:

Sample code

```java
    public void caller(int val) {

        Integer result = callRemote(val);   ①
        if (result == null) {
            System.out.println("Result is " + result);
        else {
            System.out.println("Got an exception");
        }
    }

    @CacheResult(name = "foo")
    private Integer callRemote(int val)  {

        try {
            Integer val = remoteWebServer.getResult(val);   ②
            return val;
        } catch (Exception e) {
            return null;   ③
        }
    }
```

① Call the method to call the remote

② Do the remote call and return its result

③ Return in case of exception

This approach has an unfortunate side effect: as we said before, Quarkus can also cache `null` values. Which means that the next call to `callRemote()` with the same parameter value will be answered out of the cache, returning `null` and no remote call will be done. This may be desired in some scenarios, but usually one wants to retry the remote call until it returns a result.

Let exceptions bubble up

To prevent the cache from caching (marker) results from a remote call, we need to let the exception bubble out of the called method and catch it at the caller side:

With Exception bubbling up

```
    public void caller(int val) {
        try {
            Integer result = callRemote(val);   ①
            System.out.println("Result is " + result);
        } catch (Exception e) {
            System.out.println("Got an exception");
        }
    }

    @CacheResult(name = "foo")
    private Integer callRemote(int val) throws Exception { ②

        Integer val = remoteWebServer.getResult(val);   ③
        return val;

    }
```

① Call the method to call the remote

② Exceptions may bubble up

③ This can throw all kinds of remote exceptions

When the call to the remote throws an exception, the cache does not store the result, so that a subsequent call to `callRemote()` with the same parameter value will not be answered out of the cache. It will instead result in another attempt to call the remote.

The previous code example has the side-effect that the cache logs the thrown exception with a long stack trace to the console. This is done to inform developers that something exceptional has happened, but if you have a setup like above, you are already catching the exception and know what you are doing.