

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 3.5.3+

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:1.2.1.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=cache-quickstart \
  -DclassName="org.acme.caching.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.

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/caching/WeatherForecastService.java` with the following content:

```

package org.acme.caching;

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/caching/WeatherForecast.java` this way:

```
package org.acme.caching;

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:

```

package org.acme.caching;

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) { ❶
        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);
    }
}

```

❶ 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.caching;

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. 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`.

You can only use one of the cache operations (and this annotation) on a given method: `@CacheResult`, `@CacheInvalidate` or `@CacheInvalidateAll`.

@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.

You can only use one of the cache operations (and this annotation) on a given method: `@CacheResult`, `@CacheInvalidate` or `@CacheInvalidateAll`.



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.

You can only use one of the cache operations (and this annotation) on a given method: `@CacheResult`, `@CacheInvalidate` or `@CacheInvalidateAll`.

@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.

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.



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 - ⚙️ Configuration property overridable at runtime

Configuration property	Type	Default
------------------------	------	---------

 <code>quarkus.cache.caffeine."cache-name".initial-capacity</code>	int	
 <code>quarkus.cache.caffeine."cache-name".maximum-size</code>	long	
 <code>quarkus.cache.caffeine."cache-name".expire-after-write</code>	Duration 	
 <code>quarkus.cache.caffeine."cache-name".expire-after-access</code>	Duration 	



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.

Here's what your cache configuration could look like:

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

① The `foo` cache is being configured.

② The `bar` cache is being configured.

Annotated beans examples

Implicit simple cache key

```
package org.acme.caching;

import javax.enterprise.context.ApplicationScoped;

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

@ApplicationScoped
public class CachedBean {

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

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

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

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

Explicit composite cache key

```

package org.acme.caching;

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) { ❶
        // Call expensive service here.
    }

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

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

```

- ❶ 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.caching;

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.