Quarkus - Quarkus Extension for Spring Cache API

While users are encouraged to use Quarkus annotations for caching, Quarkus nevertheless provides a compatibility layer for Spring Cache annotations in the form of the `spring-cache` extension.

This guide explains how a Quarkus application can leverage the well known Spring Cache annotations to enable application data caching for their Spring beans.

> 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
- Some familiarity with the Spring DI extension

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.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=spring-cache-quickstart \
    -DclassName="org.acme.spring.cache.GreeterResource" \
    -Dpath="/greeting" \
    -Dextensions="spring-di,spring-cache"
cd spring-cache-quickstart
```

This command generates a Maven project with a REST endpoint and imports the `spring-cache` and

`spring-di` extensions.

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

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

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

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

Creating the REST API

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

```java
package org.acme.spring.cache;

import java.time.LocalDate;

import org.springframework.stereotype.Component;

@Component
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 which contains the response sent to the users when they ask for the next three days weather forecast. Create src/main/java/org/acme/spring/cache/WeatherForecast.java this way:

```
package org.acme.spring.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.spring.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 as follows:

```java
package org.acme.cache;

import java.time.LocalDate;

import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Component;

@Component
public class WeatherForecastService {

    @Cacheable("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!

Supported features

Quarkus provides compatibility with the following Spring Cache annotations:

- `@Cacheable`
- `@CachePut`
- `@CacheEvict`

Note that in this first version of the Spring Cache annotations extension, not all features of these annotations are supported (with proper errors being logged when trying to use an unsupported feature). However, additional features are planned for future releases.

More Spring guides

Quarkus has more Spring compatibility features. See the following guides for more details:

- Quarkus - Extension for Spring DI
- Quarkus - Extension for Spring Web
- Quarkus - Extension for Spring Security
- Quarkus - Extension for Spring Data JPA
- Quarkus - Reading properties from Spring Cloud Config Server
- Quarkus - Extension for Spring Boot properties
- Quarkus - Extension for Spring Scheduled