(/)

Introduction to Spring Reactor

Last modified: April 2, 2018

by baeldung (https://www.baeldung.com/author/baeldung/)

Spring (https://www.baeldung.com/category/spring/) +

Spring 5 (https://www.baeldung.com/tag/spring-5/)

1. Overview

In this quick article, we'll introduce the Spring Reactor project. We'll set up a real-life scenario for a reactive, event-driven application.

2. The Basics of Spring Reactor

2.1. Why Reactor?

The reactive design pattern (https://en.wikipedia.org/wiki/Reactor_pattern) is an event-based architecture for asynchronous handling of a large volume of concurrent service requests coming from single or multiple service handlers.

And the Spring Reactor project is based on this pattern and has the clear and ambitious goal of building asynchronous, reactive applications on the *JVM.*

2.2. Example Scenarios

Before we get started, here are a few interesting scenarios where leveraging the reactive architectural style will make sense, just to get an idea of where you might apply it:

- Notification service of large online shopping application like Amazon
- Huge transaction processing services of banking sector
- Share trade business where share prices changes simultaneously

One quick note to be aware of is that the event bus implementation offers no persistence of events; just like the default Spring Event bus, it's an in-memory implementation.

3. Maven Dependencies

Let's start to use Spring Reactor by adding the following dependency into our *pom.xml:*

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-bus</artifactId>
    <version>2.0.8.RELEASE</version>
</dependency>
```

You can check the latest version of reactor-bus in Central Maven Repository (https://search.maven.org/classic/#search%7Cga%7C1%7Cg%3A%22io.projectreactor%22).

4. Building a Demo Application

To better understand the benefits of the reactor-based approach, **let's look at a practical example**.

We're going to build a simple notification app, which would notify users via mail and SMS – after they finish their order on an online store.

A typical synchronous implementation would naturally be bound by the throughput of the SMS service. Spikes in traffic, such holidays would generally be problematic.

With a reactive approach, the system can be more flexible and adapt better to failures or timeouts in these types of external systems, such as SMS or email servers.

Let's have a look at the application – starting with the more traditional aspects and moving on to the more reactive constructs.

4.1. Simple POJO

First, let's create a *POJO* class to represent the notification data:

```
public class NotificationData {

    private long id;
    private String name;
    private String email;
    private String mobile;

    // getter and setter methods
}
```

4.2. The Service Layer

Let's now set up a simple service layer:

```
public interface NotificationService {

    void initiateNotification(NotificationData notificationData)
      throws InterruptedException;

}
```

And the implementation, simulating a long operation here:

```
@Service
public class NotificationServiceimpl implements NotificationService {

    @Override
    public void initiateNotification(NotificationData notificationData)
      throws InterruptedException {

      System.out.println("Notification service started for "
        + "Notification ID: " + notificationData.getId());

      Thread.sleep(5000);

      System.out.println("Notification service ended for "
        + "Notification ID: " + notificationData.getId());
    }
}
```

Notice that to illustrate real life scenario of sending messages via SMS gateway or Email gateway, we're intentionally introducing a 5 seconds delay in the *initiateNotification* method by *Thread.sleep(5000).*

And so, when the thread hits the service – it will be blocked for 5 seconds.

4.3. The Consumer

Let's now jump into the more reactive aspects of our application and implement a consumer – which we'll then map to the *reactor event bus*:

```
@Service
public class NotificationConsumer implements
  Consumer<Event<NotificationData>> {

    @Autowired
    private NotificationService notificationService;

    @Override
    public void accept(Event<NotificationData> notificationDataEvent) {
        NotificationData notificationData = notificationDataEvent.getData();

        try {
            notificationService.initiateNotification(notificationData);
        } catch (InterruptedException e) {
            // ignore
        }
    }
}
```

As you can see, the consumer is simply implementing *Consumer<T>* interface – with a single *accept* method. It's this simple implementation that runs the main logic, just like a typical Spring listener (/spring-events).

4.4. The Controller

Finally, now that we're able to consume the events, let's also generate them.

We're going to do that in a simple controller:

```
@Controller
public class NotificationController {

    @Autowired
    private EventBus eventBus;

    @GetMapping("/startNotification/{param}")
    public void startNotification(@PathVariable Integer param) {
        for (int i = 0; i < param; i++) {
            NotificationData data = new NotificationData();
            data.setId(i);

            eventBus.notify("notificationConsumer", Event.wrap(data));

            System.out.println(
              "Notification " + i + ": notification task submitted successfully");
        }
    }
}
```

This is quite self-explanatory – we're sending events through the *EventBus* here – using a unique key.

So, simply put – when a client hits the URL with param value 10, a total of 10 events will be sent through the bus.

4.5. The Java Config

We're almost done; let's just put everything together with the Java Config and create our Boot application:

```
import static reactor.bus.selector.Selectors.$;

@Configuration
@EnableAutoConfiguration
@ComponentScan
public class Application implements CommandLineRunner {

    @Autowired
    private EventBus eventBus;

    @Autowired
    private NotificationConsumer notificationConsumer;

    @Bean
    Environment env() {
        return Environment.initializeIfEmpty().assignErrorJournal();
    }

    @Bean
    EventBus createEventBus(Environment env) {
        return EventBus.create(env, Environment.THREAD_POOL);
    }

    @Override
    public void run(String... args) throws Exception {
        eventBus.on($("notificationConsumer"), notificationConsumer);
    }

    public static void main(String[] args){
        SpringApplication.run(Application.class, args);
    }
}
```

It's here that we're creating the *EventBus* bean via the static *create* API in *EventBus*.

In our case, we're instantiating the event bus with a default thread pool available in the environment.

If we wanted a bit more control over the bus, we could also provide a thread count to the implementation:

```
EventBus evBus = EventBus.create(
  env,
  Environment.newDispatcher(
    REACTOR_THREAD_COUNT,REACTOR_THREAD_COUNT,
    DispatcherType.THREAD_POOL_EXECUTOR));
```

Next – also notice how we're using the static import of the *$* attribute here.

The feature provides a type-safe mechanism to include constants(in our case it's $ attribute) into code without having to reference the class that originally defined the field.

We're making use of this functionality in our *run* method implementation – **where we're registering our consumer to be triggered when the matching notification.**

This is based on **a unique selector key** that enables each consumer to be identified.

5. Test the Application

After running a Maven build, we can now simply run *java -jar name_of_the_application.jar* to run the application.

Let's now create a small JUnit test class to test the application. We would use Spring Boot's *SpringJUnit4ClassRunner* to create the test case:

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {Application.class})
public class DataLoader {

    @Test
    public void exampleTest() {
        RestTemplate restTemplate = new RestTemplate();
        restTemplate.getForObject(
          "http://localhost:8080/startNotification/10 (http://localhost:8080/startNotification/1
    }
}
```

Now, let's run this test case to test the application:

```
Notification 0: notification task submitted successfully
Notification 1: notification task submitted successfully
Notification 2: notification task submitted successfully
Notification 3: notification task submitted successfully
Notification 4: notification task submitted successfully
Notification 5: notification task submitted successfully
Notification 6: notification task submitted successfully
Notification 7: notification task submitted successfully
Notification 8: notification task submitted successfully
Notification 9: notification task submitted successfully
Notification service started for Notification ID: 1
Notification service started for Notification ID: 2
Notification service started for Notification ID: 3
Notification service started for Notification ID: 0
Notification service ended for Notification ID: 1
Notification service ended for Notification ID: 0
Notification service started for Notification ID: 4
Notification service ended for Notification ID: 3
Notification service ended for Notification ID: 2
Notification service started for Notification ID: 6
Notification service started for Notification ID: 5
Notification service started for Notification ID: 7
Notification service ended for Notification ID: 4
Notification service started for Notification ID: 8
Notification service ended for Notification ID: 6
Notification service ended for Notification ID: 5
Notification service started for Notification ID: 9
Notification service ended for Notification ID: 7
Notification service ended for Notification ID: 8
Notification service ended for Notification ID: 9
```

As you can see, as soon as the endpoint hit, all 10 tasks get submitted instantly without creating any blocking. And once submitted, the notification events get processed in parallel.

Keep in mind that in our scenario there's no need to process these events in any order.

6. Conclusion

In this small application, we definitely get a throughput increase, along with a more well-behaved application overall.

However, this scenario is just scratching the surface and represents just a good base to start understanding the reactive paradigm.

As always, the source code is available over on GitHub (https://github.com/eugenp/tutorials/tree/master/spring-reactor).

Learning to build your API

with Spring?

▲ newest ▲ oldest ▲ most voted

Guest

M Yank

really good article. one question here: on low end system, how could we assign thread pool limit?

✚ 0 ━ 2 years ago

Guest

Abhinab Kanrar

Following way you could assign thread pool:

```
@Bean
EventBus createEventBus(Environment env) {
EventBus evBus = EventBus.create(env,
Environment.newDispatcher(NUMBER_OF_THREAD_YOU_WANT_TO_ASSIGN,
NUMBER_OF_THREAD_YOU_WANT_TO_ASSIGN, DispatcherType.THREAD_POOL_EXECUTOR));
return evBus;
}
```

ideally you should assign available processor number as the thread pool size. So in the above case, NUMBER_OF_THREAD_YOU_WANT_TO_ASSIGN should be assigned in the following way:

NUMBER_OF_THREAD_YOU_WANT_TO_ASSIGN = Runtime.getRuntime().availableProcessors();

+ 1 −

2 years ago

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)
KOTLIN (HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)

THE COURSES (HTTPS://COURSES.BAELDUNG.COM)

CONSULTING WORK (/CONSULTING)

META BAELDUNG (HTTP://META.BAELDUNG.COM/)

THE FULL ARCHIVE (/FULL_ARCHIVE)

WRITE FOR BAELDUNG (/CONTRIBUTION-GUIDELINES)

EDITORS (/EDITORS)

OUR PARTNERS (/PARTNERS)

ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)

PRIVACY POLICY (/PRIVACY-POLICY)

COMPANY INFO (/BAELDUNG-COMPANY-INFO)

CONTACT (/CONTACT)