

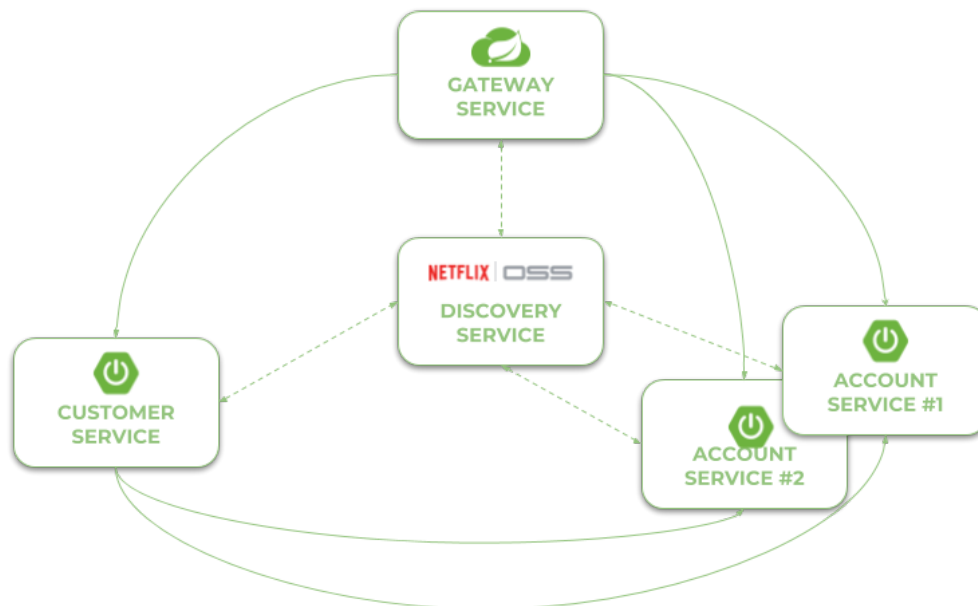
Reactive Microservices with Spring WebFlux and Spring Cloud



I have already described Spring reactive support about one year ago in the article [Reactive microservices with Spring 5](https://piotrminkowski.wordpress.com/2018/05/04/reactive-microservices-with-spring-webflux-and-spring-cloud/). At that time project **Spring WebFlux** has been under active development, and now after official release of **Spring 5** it is worth to take a look on the current version of it. Moreover, we will try to put our reactive microservices inside **Spring Cloud** ecosystem, which contains such the elements like

service discovery with **Eureka**, load balancing with Spring Cloud Commons @LoadBalanced , and API gateway using **Spring Cloud Gateway** (also based on WebFlux and Netty). We will also check out Spring reactive support for NoSQL databases by the example of **Spring Data Reactive Mongo** project.

Here's the figure that illustrates an architecture of our sample system consisting of two microservices, discovery server, gateway and MongoDB databases. The source code is as usual available on GitHub in [sample-spring-cloud-webflux](#) repository.



Let's describe the further steps on the way to create the system illustrated above.

Step 1. Building reactive application using Spring WebFlux

To enable library Spring WebFlux for the project we should include starter `spring-boot-starter-webflux` to the dependencies. It includes some dependent libraries like **Reactor** or **Netty** server.

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-webflux</artifactId>
4 </dependency>
```

REST controller looks pretty similar to the controller defined for synchronous web services. The only difference is in type of returned objects. Instead of single object we return instance of class `Mono`, and instead of list we return instance of class `Flux`. Thanks to Spring Data Reactive Mongo we don't have to do nothing more that call the needed method on the repository bean.

```
1  @RestController
2  public class AccountController {
3
4      private static final Logger LOGGER = LoggerFactory.getLogger(Acco
5
6      @Autowired
7      private AccountRepository repository;
8
9      @GetMapping("/customer/{customer}")
10     public Flux findByCustomer(@PathVariable("customer") String custo
11         LOGGER.info("findByCustomer: customerId={}", customerId);
12         return repository.findById(customerId);
13     }
14
15     @GetMapping
16     public Flux findAll() {
17         LOGGER.info("findAll");
18         return repository.findAll();
19     }
20
21     @GetMapping("/{id}")
22     public Mono findById(@PathVariable("id") String id) {
23         LOGGER.info("findById: id={}", id);
24         return repository.findById(id);
25     }
26
27     @PostMapping
28     public Mono create(@RequestBody Account account) {
29         LOGGER.info("create: {}", account);
30         return repository.save(account);
31     }
32
33 }
```

Step 2. Integrate an application with database using Spring Data Reactive Mongo

The implementation of integration between application and database is also very simple. First, we need to include starter `spring-boot-starter-data-mongodb-reactive` to the project dependencies.

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
4 </dependency>
```

The support for reactive Mongo repositories is automatically enabled after including the starter. The next step is to declare entity with ORM mappings. The following class is also returned as response by `AccountController`.

```
1 @Document
2 public class Account {
3
4     @Id
5     private String id;
6     private String number;
7     private String customerId;
8     private int amount;
9
10    ...
11
12 }
```

Finally, we may create repository interface that extends `ReactiveCrudRepository`. It follows the patterns implemented by Spring Data JPA and provides some basic methods for CRUD operations. It also allows to define methods with names, which are automatically mapped to queries. The only difference in comparison with standard Spring Data JPA repositories is in method signatures. The objects are wrapped by `Mono` and `Flux`.

```
1 public interface AccountRepository extends ReactiveCrudRepository {
2
3     Flux findByCustomerId(String customerId);
4
5 }
```

In this example I used **Docker** container for running MongoDB locally. Because I run Docker on Windows using Docker Toolkit the default address of Docker machine is **192.168.99.100**. Here's the configuration of data source in `application.yml` file.

```
1 spring:
2   data:
3     mongodb:
4       uri: mongodb://192.168.99.100/test
```

Step 3. Enabling service discovery using Eureka

Integration with Spring Cloud Eureka is pretty the same as for synchronous REST microservices. To enable discovery client we should first include starter `spring-cloud-starter-netflix-eureka-client` to the project dependencies.

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
```

Then we have to enable it using `@EnableDiscoveryClient` annotation.

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class AccountApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(AccountApplication.class, args);
7     }
8
9 }
```

Microservice will automatically register itself in Eureka. Of course, we may run more than instance of every service. Here's the screen illustrating Eureka Dashboard (<http://localhost:8761>) after running two instances of `account-service` and a single instance of `customer-service`. I would not like to go into the details of running application with embedded Eureka server. You may refer to my previous article for details: [Quick Guide to Microservices with Spring Boot 2.0, Eureka and Spring Cloud](#). Eureka server is available as `discovery-service` module.

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNT-SERVICE	n/a (2)	(2)	UP (2) - minkowp-l.p4.org:account-service:2222, minkowp-l.p4.org:account-service:2223
CUSTOMER-SERVICE	n/a (1)	(1)	UP (1) - minkowp-l.p4.org:customer-service:3333

Step 4. Inter-service communication between reactive microservices with WebClient

An inter-service communication is realized by the `WebClient` from Spring WebFlux project. The same as for `RestTemplate` you should annotate it with Spring Cloud Commons `@LoadBalanced`. It enables integration with service discovery and load balancing using Netflix OSS Ribbon client. So, the first step is to declare a client builder bean with `@LoadBalanced` annotation.

```
1  @Bean
2  @LoadBalanced
3  public WebClient.Builder loadBalancedWebClientBuilder() {
4      return WebClient.builder();
5  }
```

Then we may inject `WebClientBuilder` into the REST controller. Communication with `account-service` is implemented inside `GET /{id}/with-accounts`, where first we are searching for customer entity using reactive Spring Data repository. It returns object `Mono`, while the `WebClient` returns `Flux`. Now, our main goal is to merge those to publishers and return single `Mono` object with the list of accounts taken from `Flux` without blocking the stream. The following fragment of code illustrates how I used `WebClient` to communicate with other microservice, and then merge the response and result from repository to single `Mono` object. This merge may probably be done in more “ellegant” way, so fell free to create push request with your proposal.

```
1  @Autowired
2  private WebClient.Builder webClientBuilder;
3
4  @GetMapping("/{id}/with-accounts")
5  public Mono findByIdWithAccounts(@PathVariable("id") String id) {
6      LOGGER.info("findByIdWithAccounts: id={}", id);
7      Flux accounts = webClientBuilder.build().get().uri("http://accoun
8      return accounts
9          .collectList()
10         .map(a -> new Customer(a))
11         .mergeWith(repository.findById(id))
12         .collectList()
13         .map(CustomerMapper::map);
14  }
```

Step 5. Building API gateway using Spring Cloud Gateway

Spring Cloud Gateway is one of the newest Spring Cloud project. It is built on top of Spring WebFlux, and thanks to that we may use it as a gateway to our sample system based on reactive microservices. Similar to Spring WebFlux applications it is ran on embedded Netty server. To enable it for Spring Boot application just include the following dependency to your project.

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-gateway</artifactId>
4 </dependency>
```

We should also enable discovery client in order to allow the gateway to fetch list of registered microservices. However, there is no need to register gateway's application in Eureka. To disable registration you may set property `eureka.client.registerWithEureka` to `false` inside `application.yml` file.

```
1 @SpringBootApplication
2 @EnableDiscoveryClient
3 public class GatewayApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(GatewayApplication.class, args);
7     }
8
9 }
```

By default, Spring Cloud Gateway does not enable integration with service discovery. To enable it we should set property

`spring.cloud.gateway.discovery.locator.enabled` to `true`. Now, the last thing that should be done is the configuration of the routes. Spring Cloud Gateway provides two types of components that may be configured inside routes: **filters** and **predicates**. Predicates are used for matching HTTP requests with route, while filters can be used to modify requests and responses before or after sending the downstream request. Here's the full configuration of gateway. It enables service discovery location, and defines two routes based on entries in service registry. We use the **Path Route Predicate** factory for matching the incoming requests, and the **RewritePath GatewayFilter** factory for modifying the requested path to adapt it to the format exposed by the downstream services (endpoints are exposed under path `/`, while gateway expose them under paths `/account` and `/customer`).

```
1 spring:
```

```

2   cloud:
3     gateway:
4       discovery:
5         locator:
6           enabled: true
7     routes:
8       - id: account-service
9         uri: lb://account-service
10        predicates:
11          - Path=/account/**
12        filters:
13          - RewritePath=/account/(?.*), /$\{path}
14       - id: customer-service
15         uri: lb://customer-service
16        predicates:
17          - Path=/customer/**
18        filters:
19          - RewritePath=/customer/(?.*), /$\{path}

```

Step 6. Testing the sample system

Before making some tests let's just recap our sample system. We have two microservices `account-service`, `customer-service` that use MongoDB as a database. Microservice `customer-service` calls endpoint `GET /customer/{customer}` exposed by `account-service`. URL of `account-service` is taken from **Eureka**. The whole sample system is hidden behind gateway, which is available under address **localhost:8090**. Now, the first step is to run MongoDB on Docker container. After executing the following command Mongo is available under address **192.168.99.100:27017**.

```

1 | $ docker run -d --name mongo -p 27017:27017 mongo

```

Then we may proceed to running `discovery-service`. Eureka is available under its default address **localhost:8761**. You may run it using your IDE or just by executing command `java -jar target/discovery-service-1.0-SNAPSHOT.jar`. The same rule applies to our sample microservices. However, `account-service` needs to be multiplied in two instances, so you need to override default HTTP port when running second instance using `-Dserver.port` VM argument, for example `java -jar -Dserver.port=2223 target/account-service-1.0-SNAPSHOT.jar`. Finally, after running `gateway-service` we may add some test data.

```

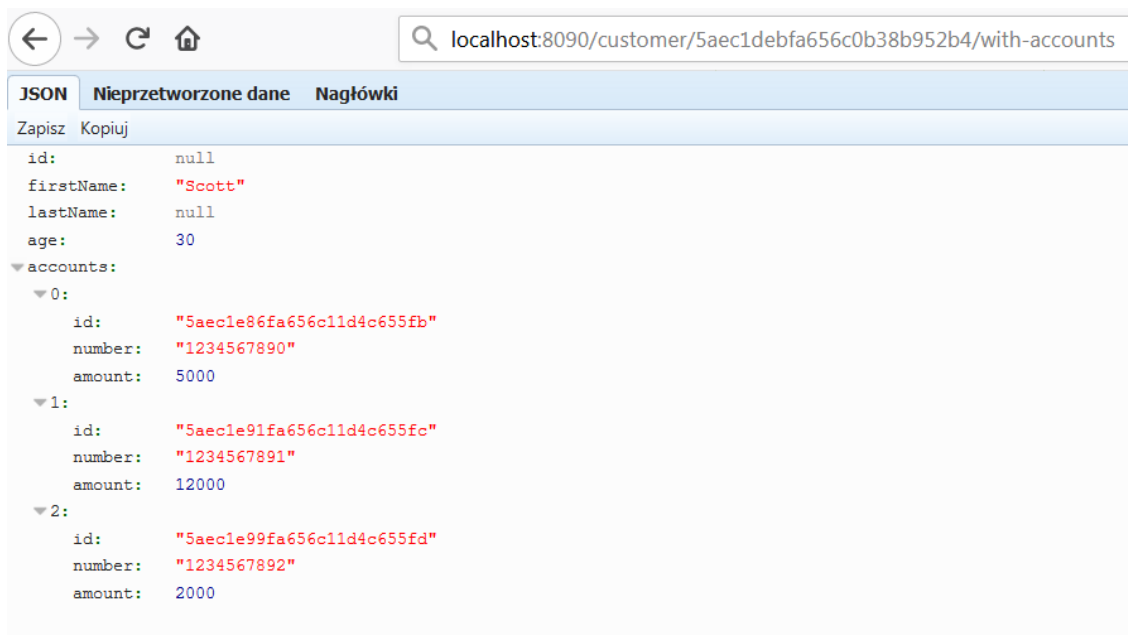
1 | $ curl --header "Content-Type: application/json" --request POST --data
2 | {"id": "5aec1debfa656c0b38b952b4", "firstName": "John", "lastName": "Sco

```



```
3 $ curl --header "Content-Type: application/json" --request POST --data
4 {"id": "5aec1e86fa656c11d4c655fb", "number": "1234567892", "customerId":
5 $ curl --header "Content-Type: application/json" --request POST --data
6 {"id": "5aec1e91fa656c11d4c655fc", "number": "1234567892", "customerId":
7 $ curl --header "Content-Type: application/json" --request POST --data
8 {"id": "5aec1e99fa656c11d4c655fd", "number": "1234567892", "customerId":
```

To test inter-service communication just call endpoint `GET /customer/{id}/with-accounts` on `gateway-service`. It forward the request to `customer-service`, and then `customer-service` calls endpoint exposed by `account-service` using reactive `WebClient`. The result is visible below.



Conclusion

Since Spring 5 and Spring Boot 2.0 there is a full range of available ways to build microservices-based architecture. We can build standard synchronous system using **one-to-one** communication with Spring Cloud Netflix project, **messaging** microservices based on message broker and publish/subscribe communication model with Spring Cloud Stream, and finally asynchronous, **reactive** microservices with Spring WebFlux. The main goal of this article is to show you how to use Spring WebFlux together with Spring Cloud projects in order to provide such a mechanisms like service discovery, load balancing or API gateway for reactive microservices build on top of Spring Boot. Before Spring 5 the lack of support for reactive microservices

was one of the drawback of Spring framework, but now with Spring WebFlux it is no longer the case. Not only that, we may leverage Spring reactive support for the most popular NoSQL databases like MongoDB or Cassandra, and easily place our reactive microservices inside one system together with synchronous REST microservices.

Advertisements



I doubled my earnings with

WordAds

[LEARN MORE >](#)

[REPORT THIS AD](#)

Earn money from your WordPress site

WordAds



[REPORT THIS AD](#)

Share this:

 Facebook 1

 More

Like



2 bloggers like this.

Related



[Quick Guide to Microservices with Spring Boot 2.0, Eureka and Spring Cloud](#)

In "microservices"



[The Future of Spring Cloud Microservices After Netflix Era](#)

In "microservices"



[Introduction to Reactive APIs with Postgres, R2DBC, Spring Data JDBC and Spring WebFlux](#)

In "other"



Author: Piotr Mińkowski

IT Architect, Java Software Developer [View all posts by Piotr Mińkowski](#)



Piotr Mińkowski / May 4, 2018 / microservices / microservices, MongoDB, netty, reactive, Reactor, spring boot, spring cloud, spring-webflux

4 thoughts on “Reactive Microservices with Spring WebFlux and Spring Cloud”



Sridhar

May 6, 2018 at 10:54 am

Hi,

Thanks for sharing an informative article on reactive micro-services.

I have a few questions as below.

Is it a best practice to specifically add `@EnableDiscoveryClient` on a service because, by default (In latest versions of Spring Boot/Cloud) when cloud dependencies are in the classpath the service is implicitly a eureka client.

Also, instead of using `WebClient.Builder` can we use Spring Cloud Open Feign as suggested by one of your earlier articles to consume one reactive service from another. If yes, is there a scenario where we need to use one over the other.

★ Like



Piotr Mińkowski 👤

May 6, 2018 at 5:02 pm

Hi,

It's really interesting question. Following this discussion it should be possible with Feign client: <https://github.com/spring-cloud/spring-cloud-netflix/issues/1554>. However, I'm not sure and I didn't try it. Probably it is still before acceptance following this link: <https://github.com/spring-cloud/spring-cloud-openfeign/issues/4>.

And you are right with Eureka client. I did not figure out that in the latest version of Spring Cloud (Finchley.RC1) `@EnableDiscoveryClient` is not required. Thanks 😊

★ Like



Marco Giglione

July 13, 2018 at 9:44 am

Hi Piotr,

I wrote a post focused on code of a simple reactive demo application. I think that it could be useful for beginners to understand the subject. With your permission I'll share the url <http://www.mgiglione.com/2018/05/23/reactive-webflux/>



Piotr Mińkowski

July 13, 2018 at 11:49 am

Hi Marco,

Ok 😊



This site uses Akismet to reduce spam. [Learn how your comment data is processed.](#)

Piotr's TechBlog / [Create a free website or blog at WordPress.com.](#)