[[Reactive Spring]]

Reactive Spring

by Josh Long (@starbuxman), Spring Developer Advocate, Pivotal

The Need for Async IO

At Pivotal we see a lot of organizations moving to microservices. The architecture emphasizes small, singly focused, independently deployable pieces of functionality. They're small because small implies a smaller team working on the codebase. They're singly focused because it's easier to stay small that way. They're independently deployable because that means you're never blocked by other teams in the organization. The architecture has a lot of benefits; it supports organizational agility. But, the approach implies network distribution. The distribution in turn invites architectural complexity. Things get difficult when dependencies and networks fail. Things get difficult when the conduction of data between nodes becomes overwhelming.

Blocking IO makes the situation worse. When we say blocking, we mean that for a client to read a `byte` from a file or a socket (or whatever), it needs to wait for the `byte` to arrive. The thread in which the read is happening is idle, but unusable. On the JVM, where creating threads is very expensive, having wasted resources like this is all the more the pity. It doesn't need to be this way. At the lower levels in an operating system, network and file IO is typically asynchronous by default.

Java has always had `java.io.InputStream` and `java.io.OutputStream`, which are blocking abstractions. Java 1.4 saw the introduction of NIO (`java.nio.*`), the *new* IO packages. NIO supports asynchronous IO - it provides a way to open a resource and be notified when there is new data, while not blocking the reading thread. This is a marked improvement but you'll notice that it hasn't exactly become the pervasive metaphor for computation. These days almost all of the APIs typical of enterprise computing are blocking by default and you probably don't even realize it. Hopefully, most of us aren't writing data access layers, HTTP web stacks, and security integrations ourselves. We're relying on tried and true implementations that sit below the value line, below the line of things that we feel the need to expend cognitive effort trying to understand. Our business application is what matters, and so most frameworks let us think about the world in terms of our business domain, relegating IO concerns to the murky depths of the of the framework, the data access layer, etc.

We think about the world in terms of instances of our domain model entities, `T`, or collections of `T`. They are synchronous views on data. The `CompletableFuture<T>` gives us an asynchronous views of a single value. It lets us say that, *eventually*, we'll have a reference to a `T`. We haven't had for the longest time a way to say that we have a potentially infinite collection of type `T` that may or may not arrive, *eventually*.

Such a metaphor would need to be coupled with a way to gate the production and consumption of values. In the world of finite data sets, producers may occasionally overwhelm consumers but a consumer can buffer extra work, allowing it to eventually get to it. This works because the buffer is sufficient to absorb the overwhelming demand. What happens if the production is infinite? Eventually, like occupants in a leaky boat desperately trying to bail out the ocean, the consumer

will sink. This is called flow control, and it needs to be a first class feature of any computational metaphor we choose.

The Missing Metaphor

We haven't had a good metaphor to address this use case. Not for want of trying, of course... Many have done a lot of great work in this space. Microsoft Research kicked things off with the RX extensions for .NET. Netflix ported that to Java and created RX Java. RX Java has in turn inspired countless clones in other languages and platforms like Swift, Node.js, and Ruby. The Spring team launched a project called Reactor. Lightbend (ne' Typesafe) tried to support this space in Akka and the Akka Streamsproject. Tim Fox, first at VMWare, and then at RedHat, launched the vert.x project. all of these attempt to address the same usecases: fill in the gap in our idioms and then, tow whatever extn possible, develop an ecosyste of tools on top of the new idiom that suppots modern application development concerns.

The Reactive Streams Initiative

There's common ground across these different approaches. The aforementioned vendors worked together to develop a de-facto standard, the Reactive Streams initiative. The Reactive Streams initiative defines four types:

The `Publisher<T>` is the computational metaphor we've been looking for. It defines the missing piece; a producer of values that may eventually arrive. A `Publisher<T>` produces values of type `T`.

Example 1. the Reactive Streams `Publisher<T>`.

```
package org.reactivestreams;

public interface Publisher<T> {

    void subscribe(Subscriber<? super T> s);
}
```

The `Subscriber` subscribes to a `Publisher<T>`, receiving notifications on any new values of type `T`.

Example 2. the Reactive Streams `Subscriber<T>`.

```
package org.reactivestreams;

public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable t);

    public void onComplete();
}
```

When a `Subscriber<T>` subscribes to a `Publisher<T>`, it results in a `Subscription<T>`.

Example 3. The Reactive Streams `Subscription<T>`.

```
package org.reactivestreams;

public interface Subscription {

    public void request(long n);

    public void cancel();
}
```

A `Publisher<T>` that is also a `Subscriber<T>` is called a `Processor<T>`.

Example 4. The Reactive Streams `Processor<T>`.

```
package org.reactivestreams;

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

The `Publisher<T>` has one method, `Publisher#subscribe(Subscriber<T>)`. The `Subscriber` has a callback method, `Subscriber#onSubscribe(Subscription)`. that is invoked by the `Publisher<T>`. This is the first and last interaction between the `Publisher<T>` and the `Subscriber<T>` *until* the subscriber calls the `Subscription#request(long numberOfRecordsRequested)` method. Here, the `Subscriber<T>` signals to the `Publisher<T>` how many more records, `T`, it is prepared to handle. The `Subscriber<T>` can not be overwhelmed - it will receive only as many records as it can handle. The `Subscriber#onNext(T)` method is be called only as many times as requested. This dynamic - where

the producer gates the production based on capacity - is called *backpressure*; it's a way of managing flow.

The Reactive Streams specification serves as a foundation for interoperability. It provides a common way to address this missing metaphor. The specification is not mean to be a prescription for the implementation APIs, it instead defines types for interoperability.

The Reactive Streams types eventually found their way into Java 9 as one to one semantically equivalent interfaces in the `java.util.concurrent.Flow` class. The Reactive Streams initiative, for the moment, remains a separate set of types but the expectation is that all implementations will also support Java 9 as soon as possible.

Reactor

Are the types in the Reactive Streams initiative enough? I'd say *no*! The Reactive Streams specification is a bit like a plain vanilla Java array: it provides a basis on which to support higher order computations. Most of us don't use `T[]` arrays directly, we use something that extends from `java.util.Collection<T>`. The same is true for the basic Reactive Streams types. In order to support filtering, transforming, and iteration, you'll need DSLS and that's where Reactive Streams implementations can really help.

Pivotal's Reactor project is a good choice here; it's built on top of the Reactive Streams specification. It provides two specializations of the `Publisher<T>`. The first, `Flux<T>`, is a Publisher that produces zero or more values. It's unbounded. The second, `Mono<T>`, is a `Publisher<T>` that produces zero or one value. They're both publishers and you can treat them that way, but they go much further than the Reactive Streams specification. They both provide operators, ways to process a stream of values. Reactor types compose nicely - the output of one thing can be the input to another.

Reactive Spring

As useful as project Reactor is, it's only a foundation. Our applications need to talk to data sources. They need to produce and consume HTTP, SSE and WebSocket endpoints. They support authentication and authorization. Spring provides these things. If Reactor gives us the missing metaphor, Spring helps us all speak the same language.

Spring Framework 5.0 was released in September 2017. It builds on Reactor and the Reactive Streams specification. It includes a new reactive runtime and component model called Spring WebFlux. Spring WebFlux does not depend on or require the Servlet APIs to work. It ships with adapters that allow it to work on top of a Servlet-engine, if need be, but it's not required. It also provides a Netty-based web server. Spring Framework 5, which works with a baseline of Java 8 and Java EE 7 - is the foundation for changes in much of the Spring ecosystem.

A Bootiful Application

Let's look at an example. We'll build a simple Spring Boot 2.0 application that, oh, I don't know… How about we build a service to manage books? We could call the project *Library* or something like that.

The Spring Initializr

Go to the Spring Initializr. Make sure that some version of Spring Boot 2.0 (or later) is selected in the version drop down menu. We're writing a service to manage access to books in the library, so give this project the artifact ID `library-service`. Select `Reactive Web`, `Actuator`, `Reactive MongoDB`, `Reactive Security`, and `Lombok`. I chose to use the Kotlin language, even if most of the project we'll build is in Java. You can keep Java artifacts in a Kotlin project. Then, click *Generate* and it'll download an archive; unzip it and open it in your favorite IDE that supports Java 8 (or later), Kotlin (optionally) and Maven. While we could've chosen Gradle at the Spring Initializr, I've chosen Maven for the purposes of this article.

Our stock standard Spring Boot application has an entry class that looks like this.

Example 5. the empty husk of a new Spring Boot project.

```
package com.example.libraryservice;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class LibraryServiceApplication {

    public static void main(String[] args) {
        // todo
        System.setProperty("spring.profiles.active", "security,authorization,frp-
java");
        SpringApplication.run(LibraryServiceApplication.class, args);
    }
}
```

Data Access with Reactive Spring Data modules

Spring Data Kay is the largest update to Spring Data since its inception. This release debuts support, where supported in the underlying data stores (MongoDB, Cassandra, Redis and Couchbase), for reactive data access. It introduces new reactive repository and template implementations. We've got the reactive MongoDB driver and Spring Data module on the classpath so let's use them to manage some data. Create a new entity called `Book`.

Example 6. a MongoDB `@Document` entity, `Book`

```
package com.example.libraryservice;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Book {
    @Id
    private String id;
    private String title;
    private String author;
}
```

Create a Spring Data repository to support the data management lifecycle of the entity. This should look very familiar for anybody who's ever used Spring Data, except that the repository supports *reactive* interactions: methods return `Publisher` types, and input can be given as `Publisher<T>` instances.

Example 7. A reacive Spring Data MongoDB repository

```
package com.example.libraryservice;

import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import reactor.core.publisher.Flux;

public interface BookRepository extends ReactiveMongoRepository<Book, String> {

    Flux<Book> findByAuthor(String author);
}
```

Install Some Sample Data with an `ApplicationRunner`

With that we have enough to install some sample data (just for our demo). Spring Boot invokes the `#run(ApplicationArguments)` method when the application has started, passing in wrappers for the arguments (`String [] args`) into the application. Let's create an `ApplicationRunner` that deletes all the data in the data source, then emits a few book titles, then maps them to `Book` entities, and then

persists those books; then queries all the records in the data source and then prints out everything.

Example 8. an `ApplicationRunner` to write data

```
package com.example.libraryservice;

import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Flux;
import reactor.core.scheduler.Scheduler;
import reactor.core.scheduler.Schedulers;

import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ThreadFactory;
import java.util.function.Supplier;

@Slf4j
@Component
class SampleBookInitializer implements ApplicationRunner {

    private final BookRepository bookRepository;

    SampleBookInitializer(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @Override
    public void run(ApplicationArguments args) throws Exception {

        // @formatter:off
        this.bookRepository
            .deleteAll()
            .thenMany(
              Flux.just("Professional Java Development with the Spring
Framework|rjohnson",
                "Cloud Native Java|jlong", "Spring Security 3.1|rwinch", "Spring
in Action|cwalls"))
            .map(t -> t.split("\\|"))
            .map(tuple -> new Book(null, tuple[0], tuple[1]))
            .flatMap(this.bookRepository::save)
            .thenMany(this.bookRepository.findAll())
            .subscribe(book -> log.info(book.toString()));
        // @formatter:on
    }
}
```

The example looks at the titles of various books, and one of (the possibly numerous) book's authors,

and writes them to the database. First the strings are split by the `|` delimiter. Then the title and book author are used to create a `Book`. Then the records are saved to the data source, MongoDB. The result of the `save` operation is a `Mono<Book>`. Something needs to subscribe to each of those resulting `Publisher<T>`s, so we use the `flatMap` operator. Then, we switch tracks, turning focusing on the results of finding all records and then logging them for inspection.

This code defines a pipeline; each operator defines a stage in a pipeline. The pipeline is not *hot*, or *eager*. We need to activate the pipeline by subscribing to it. The `Publisher<T>` only defines one type of subscription, but Reactor defines a few overloads that provide the same lifecycle hooks as overriding individual methods in a fully formed `Subscriber<T>` would. There are hooks to process each emitted value, as well any exceptions thrown, among other things.

Reactive and in a Rush? Use a `Scheduler`!

Were you to put a breakpoint in any of the lambdas in the previous code listing and then inspect `Thread.currentThread().getName()`, you'd see that the the thread on which processing is running is different than the main thread (which is named `main`). Reactor defers to a `Scheduler` implementation for its processing. You can specify the default global `Scheduler` you'd like to use by calling `Schedulers.setFactory(Factory)`. You can specify on which thread a particular `Publisher<T>` should run when it subscribes by specifying `Mono<T>#subscribeOn(Scheduler)` or `Flux<T>#subscribeOn(Scheduler)`.

Spring WebFlux

Now that we've got data in the data source, let's stand up a REST API. We'll use Spring WebFlux, a brand new reactive web runtime and component model. Spring WebFlux does not depend on the Servlet specification. It can work independently, with a Netty-based web server. It is designed, from the bottom up, to work with `Publisher<T>` instances.

REST with Spring WebFlux

We can use Spring MVC style controllers, like this:

Example 9. a Spring MVC style REST API.

```
package com.example.libraryservice;

import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;

@Profile("mvc-style")
@RestController
class BookRestController {

    private final BookRepository bookRepository;

    BookRestController(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    @GetMapping("/books")
    Flux<Book> all() {
        return this.bookRepository.findAll();
    }

    @GetMapping("/books/{author}")
    Flux<Book> byAuthor(@PathVariable String author) {
        return this.bookRepository.findByAuthor(author);
    }
}
```

NOTE

Spring has the concept of profiles. Profiles are labels, or tags, essentially, for Spring beans. Beans in a given profile don't exist unless that profile is activated. The easiest way to activate a profile is to use a command line argument when running the `java` command. If you wanted to activate all the beans under the `profile1` and `profile2` profiles, you'd use an incantation like this: `java -Dspring.profiles.active=profile1,profile2 -jar ...`. The benefit of the profile, in this case, is that we can have the same HTTP endpoints implemented three different ways in the same codebase and only activate one at a time.

This controller should look familiar to anyone who's ever used Spring MVC. It may be familiar, but it is *not* Spring MVC. We're using a new reactive runtime called Spring WebFlux. The annotations are the same, but the rules are sometimes different.

Functional Reactive Endpoints

The last code listing demonstrates a controller. Spring Web Flux controllers define endpoint handlers and endpoint mappings through declarative annotations. The annotations describe how the routing for a given endpoints is to be handled. They're sophisticated, but ultimately limited to whatever the framework itself can do with those annotations. If you want more flexible request matching capabilities, you can use Spring WebFlux functional reactive endpoints.

Example 10. The same endpoints reworked as funtional reactive endpoints in Java. Run this using the `frp-java` profile.

```
package com.example.libraryservice;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.web.reactive.function.server.RouterFunction;

import static
org.springframework.web.reactive.function.server.RequestPredicates.GET;
import static
org.springframework.web.reactive.function.server.RouterFunctions.route;
import static org.springframework.web.reactive.function.server.ServerResponse.ok;

@Profile("frp-java")
@Configuration
class BookRestConfigurationJava {

    //@formatter:off
    @Bean
    RouterFunction<?> routes(BookRepository br) {
        return
            route(GET("/books"),
                req -> ok().body(br.findAll(), Book.class))
            .andRoute(GET("/books/{author}"),
                req -> ok().body(br.findByAuthor(req.pathVariable("author")),
Book.class));
    }
    //@formatter:on
}
```

The functional reactive style lets you express HTTP endpoints as request predicates mapped to a handler class. The handler class implementation is easily expressed as concise Java lambdas. You can use the default request predicates or provide your to gain full control over how requests are matched and dispatched.

We produce a result and pass it to the `body(Publisher<T>)` method, along with a class literal. We need the class literal to help the engine figure out what type of message framing it should do.

Remember, that `Publisher<T>` *might* produce a TRILLION records! It might not ever stop! The producer can't afford to wait until all records have been produced and *then* marshal the record from an object to JSON. So, it marhsals each record as soon as it's got it. We need to tell it what kind of message to look out for. In the Spring MVC style controllers, the return value (a `Publisher<T>`) in the handler methods encodes its generic parameter, `T`, and the engine can retrieve that generic parameter using reflection. The engine can *not* do the same thing for the instance variable passed into the `body` method as a parameter since there's no easy way to retrieve the generic signature of instance variables. We call this limitation *type erasure*. The type literal gets us past this restriction.

If you're using the Kotlin language, things are even more concise thanks to a Kotlin-language DSL that also ships as part of Spring Framework 5. The Kotlin DSL requires less code and also supports retrieving the generic parameter thanks to runtime reification of inline methods. Here are the same endpoints reimplemented using the Kotlin-language DSL:

Example 11. Here are the same endpoints using the Kotlin-language DSL

```
package com.example.libraryservice

import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile
import org.springframework.web.reactive.function.server.ServerResponse.ok
import org.springframework.web.reactive.function.server.body
import org.springframework.web.reactive.function.server.router

@Profile("frp-kotlin")
@Configuration
class BookRestConfigurationKotlin {

    //@formatter:off
    @Bean
    fun routes(br: BookRepository) = router {
        GET("/books") { r -> ok().body(br.findAll()) }
        GET("/books/{author}") { r ->
ok().body(br.findByAuthor(r.pathVariable("author"))) }
    }
    //@formatter:on

}
```

Observability with Spring Boot 2.0's Actuator

Alright! Not bad. We've got a REST API. (Three, even!) Can we go to production yet? Not yet. There are a number of concerns that need to be addressed first. One is observability. Observability is the insight that, no matter the technical stack, business vertical, geography or deployment model, one

thing is surprisingly consistent: when the pager (or PagerDuty) goes off, *somebody* gets roused to a computer and has to get things back online. *A human being*.

The goal is not root cause analysis but instead to salve the wound. When the system is down, every second counts and we need to support the recovery process with as much visibility into the system as possible. Different levels of the runtime will aid you here: a smart platform, like Cloud Foundry, can tell us a lot about the container running the application. The application itself is in the best situation to articulate its own state. The Spring Boot Actuator can help us here.

The Spring Boot Actuator isn't uniquely reactive. It's been in Spring Boot from the beginning. The news here is that, as of Spring Boot 2.0, it has been divorced from any particular web runtime. It no longer requires Spring MVC; it can work with Spring WebFlux, Spring MVC, or JAX-RS. Specify that it export the HTTP web endpoints with a bit of configuration:

Example 12. configuration to expose the Actuator endpoints for http access

```
endpoints.default.web.enabled=true
```

All the Actuator endpoints are in place, contextualized under `/application`. There are a number of endpoints including `/application/trace`, `/application/health`, `/application/env`, and `/application/metrics`.

Example 13. The output of the `/application/health` endpoint.

```
{
   "status" : "UP",
   "details" : {
      "diskSpace" : {
         "status" : "UP",
         "details" : {
            "free" : 685532655616,
            "total" : 1004314628096,
            "threshold" : 10485760
         }
      },
      "mongo" : {
         "details" : {
            "version" : "3.2.11"
         },
         "status" : "UP"
      }
   }
}
```

The metrics endpoint has been reworked in Spring Boot 2.0. It's now backed by a new project called Micrometer. Micrometer is an abstraction for dimensional metrics publication and accumulation. It

integrates with time series databases like Prometheus, Netflix Atlas, and Datadog. Support for InfluxDB, statsd, and Graphite are coming. Spring Boot captures metrics related to the application, and you can capture custom metrics as well.

Example 14. The output of the `/application/metrics` endpoint.

```
{
   "names" : [
      "jvm.buffer.memory.used",
      "jvm.memory.used",
      "jvm.buffer.count",
      "logback.events",
      "process.uptime",
      "jvm.memory.committed",
      "http.server.requests",
      "jvm.buffer.total.capacity",
      "jvm.memory.max",
      "process.starttime"
   ]
}
```

Spring Security

I know what you're thinking, but its **still** not quite ready for production. We need to address security. We'll use Spring Security 5.0. Spring Security supports a rich set of integrations with all manner of identity providers. It supports authentication and authorization. Spring Security supports authentication by propagating a security context so that application level code - method invocations, HTTP requests, etc. - have easy access to the context. The context has historically been implemented with a `ThreadLocal`. This makes a lot of sense in a non-reactive world, but less so in a reactive world. Reactor provides a `Context` object, which acts as a sort of dictionary. Spring Security 5.0's reactive support propagates its security context using this mechanism. Parallel, reactive type hierarchies have been introduced to support non-blocking authentication and authorization.

You don't have to worry about much of this nuance. I just think it's dope. All we need to know is that in the reactive world, authentication is handled by an object of type `ReactiveAuthenticationManager` which has a simple job: given an `Authentication` attempt, return a `Mono<Authentication>` (indicating whether the authentication attempt succeeded) or throw an `Exception`.

One implementation of the `ReactiveAuthenticationManager` supports delegating to a user-provided object of type `UserDetailsRepository`. The `UserDetailsRepository` connects your custom username and password store to Spring Security's authentication. You might have a database table called `USERS` or just a hardcoded `Map<K,V>` of users.

By default, Spring Security locks down the whole application and installs HTTP BASIC authentication. Any attempt at calling any endpoint will fail unless we provide credentials. By default all authenticated principals can access all endpoints.

Let's establish introduce a few users of various roles. All users will have the `USER` role, but only a privileged few will have the `ADMIN` role. In this newly secured world, let's say that all users will be able to view the books they've written, but only those with the `ADMIN` role will be able to see *all* the books. (Let's ignore for now whether this domain makes any sense!)

Example 15. The Spring Security configuration

```
package com.example.libraryservice;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.security.authorization.AuthorizationDecision;
import
org.springframework.security.config.annotation.web.reactive.EnableWebFluxSecurity;
import org.springframework.security.config.web.server.HttpSecurity;
import org.springframework.security.core.userdetails.MapUserDetailsRepository;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.server.SecurityWebFilterChain;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

@Profile("security")
@Configuration
@EnableWebFluxSecurity
class SecurityConfiguration {

    @Bean
    MapUserDetailsRepository authentication() {
        return new MapUserDetailsRepository(
                user("rjohnson", "ADMIN"),
                user("cwalls"),
                user("jlong"),
                user("rwinch", "ADMIN"));
    }

    //@formatter:off
    @Bean
    @Profile("authorization")
    SecurityWebFilterChain authorization(HttpSecurity http) {
        return
            http
                .httpBasic()
                    .and()
                .authorizeExchange()
                        .pathMatchers("/books/{author}").access((auth, ctx) ->
                            auth
```

```
                            .map(authentication -> {
                                Object author = ctx.getVariables().get("author");
                                boolean matchesAuthor =
authentication.getName().equals(author);
                                boolean isAdmin =
authentication.getAuthorities().stream()
                                        .anyMatch(ga ->
ga.getAuthority().contains("ROLE_ADMIN"));
                                return (matchesAuthor || isAdmin);
                            })
                            .map(AuthorizationDecision::new)
                        )
                        .anyExchange().hasRole("ADMIN")
                    .and()
                .build();
    }
    //@formatter:on

    private static UserDetails user(String u, String... roles) {
        List<String> r = new ArrayList<>(Arrays.asList(roles));
        r.add("USER");
        String[] rolesArray = r.toArray(new String[0]);
        return User.withUsername(u).password("pw").roles(rolesArray).build();
    }
}
```

Let's try making an HTTP BASIC authenticated call to the service. I'll make the first request as `jlong`, a regular `USER`.

Example 16. `curl` the endpoint as `jlong`

```
curl -ujlong:pw http://localhost:8080/books/jlong
```

It won't work if I try to access `http://localhost:8080/books/rwinch`, though. Only `ADMIN` role users can access other endpoints.

Example 17. `curl` the endpoint as `rwinch`

```
curl -urwinch:pw http://localhost:8080/books
```

That will work. We've only begun to scratch the surface of application security.

Deployment

Our application is secure and observable. *Now* we can deploy it. This is a natural thing to run in a cloud provider like Cloud Foundry, an open-source Apache 2 licensed cloud platform that's optomized for the continuous management of applications. It sits at a level (or two) above cloud infrastructure. It is infrastructure agnostic, running on local cloud providers like OpenStack and vSphere or on public cloud providers like Amazon Web Services, Google Cloud, Microsoft Azure and, yes, [Oracle Cloud](https://blogs.oracle.com/developers/cloud-foundry-arrives-on-oracle-cloud)! No matter where Cloud Foundry is installed, it's use is basically the same. You authenticate and then tell the platform about your application workload using the `cf` CLI and the `cf push` command.

Example 18. Using `cf` CLI to push the application.

```
cf login -a $CF_API_ENDPOINT -u $CF_USER -s $CF_SPACE -o $CF_ORG

cf push -p library-service-0.0.1-SNAPSHOT.jar java-magazine-library-service
```

Once the application is up and running you can access its public HTTP endpoints. You can provision backing services - message queues, databases, caches, etc. - using `cf create-service`. You can scale the application up to multiple load-balanced instances using `cf scale`. You can interrogate the application's metrics, its Spring Boot Actuator endpoints, its health and so much more, all from the Pivotal Apps Manager dashboard. The application is up and running and our clients can talk to it in a secure fashion.

"Wait, what client?," I hear you saying...

A (Reactive) Client

We've stood up a REST API. We need to connect a client to the service. While we could use the Spring Framework `RestTemplate`, the general workhorse HTTP client that has served us well for the better part of a decade, it's not particularly suited to potentially unlimited streams of data. It expects to be able to convert all response payloads into something by waiting for the end of the response. This isn't going to work if the client is using server sent events, or even just a really large JSON response. Instead, let's use the new Spring WebFlux `WebClient`.

Example 19. Configuring and using an authenticated `WebClient`

```
package com.example.libraryclient;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.web.reactive.function.client.ExchangeFilterFunction;
import org.springframework.web.reactive.function.client.ExchangeFilterFunctions;
import org.springframework.web.reactive.function.client.WebClient;

@SpringBootApplication
public class LibraryClientApplication {

    @Bean
    WebClient client(@Value("${libary-service-url:http://localhost:8080/}") String
url) {
        ExchangeFilterFunction basicAuth = ExchangeFilterFunctions
                .basicAuthentication("rwinch", "pw");
        return WebClient.builder().baseUrl(url).filter(basicAuth).build();
    }

    @Bean
    ApplicationRunner run(WebClient client) {
        //@formatter:off
        return args ->
            client
                .get()
                .uri("/books")
                .retrieve()
                .bodyToFlux(Book.class)
                .subscribe(System.out::println);
        //@formatter:on
    }

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

@Data
@AllArgsConstructor
@NoArgsConstructor
class Book {
    private String id;
    private String title;
    private String author;
}
```

We configure the `WebClient` and pre-configure a `baseUrl` as well as an `ExchangeFilterFunction` that authenticates the client with the service. The `WebClient` gives us a `Publisher<T>` for the response, which we then print the console. In this case, it doesn't really matter; we've only got four records in our endpoint! The thing that's so valuable is that this code would work even if we were consuming a trillion records. It would work if we were consuming server-sent events that never end.

Subscriber#onNext

What's next? In this article we've looked briefly at building a web service with Spring Boot. We looked at Reactor, Spring Data Kay, Spring Framework 5 and Spring WebFlux, Spring Security 5, and Spring Boot 2.0.

Spring Boot 2 sets the stage for Spring Cloud Finchley. Spring Cloud Finchley builds on Spring Boot 2.0 and updates a number of different APIs, where appropriate, to support the reactive paradigm. Service registration and discovery works in Spring WebFlux based applications. Spring Cloud Commons supports client-side load-balancing across services registered in a service registry (like Apache Zookeepere, Hashicorp Consul and Netflix Eureka) for the Spring Framework `WebClient`. Spring Cloud Netlix Hystrix circuit breakers have always worked naturally with RxJava, which in turn can interop with `Publisher<T>` instances. This is even easier. Spring Cloud Stream supports working with `Publisher<T>` instances to describe how messages arrive and are sent to messaging subsraits like RabbitMQ, Apache Kafka or Redis. Spring Cloud Gateway is a brand new reactive API gateway project that supports HTTP and websocket request proxying, rewriting, load-balancing, circuit breaking, rate limiting and so much more. Spring Cloud Sleuth has been updated to support distributed tracing across reactive boundaries. The list goes on and on.

The `Future<Spring>` is a reactive `Publisher<Spring>`. Begin your journey building production-worthy, agile and reactive applications and services with Spring Boot at the Spring Initializr. If you have questions, find me on Twitter @starbuxman or email (josh@joshlong.com).