

Spring MVC

jim stafford

Fall 2022 v2022-07-15: Built: 2022-12-07 06:12 EST

Table of Contents

1. Introduction	1
1.1. Goals	1
1.2. Objectives	1
2. Spring Web APIs	2
2.1. Lecture/Course Focus	2
2.2. Spring MVC	2
2.3. Spring WebFlux	3
2.4. Synchronous vs Asynchronous	3
2.5. Mixing Approaches	4
2.6. Choosing Approaches	5
3. Maven Dependencies	6
4. Sample Application	7
5. Annotated Controllers	8
5.1. Class Mappings	8
5.2. Method Request Mappings	9
5.3. Default Method Response Mappings	9
5.4. Executing Sample Endpoint	10
6. RestTemplate Client	12
6.1. JUnit Integration Test Setup	12
6.2. Form Endpoint URL	13
6.3. Obtain RestTemplate	14
6.4. Invoke HTTP Call	14
6.5. Evaluate Response	15
7. WebClient Client	16
7.1. Obtain WebClient	16
7.2. Invoke HTTP Call	16
8. Implementing Parameters	18
8.1. Controller Parameter Handling	18
8.2. Client-side Parameter Handling	19
9. Accessing HTTP Responses	20
9.1. Obtaining ResponseEntity	20
9.2. ResponseEntity<T>	20
10. Client Error Handling	22
10.1. RestTemplate Response Exceptions	22
10.2. WebClient Response Exceptions	23
10.3. RestTemplate and WebClient Exceptions	23
11. Controller Responses	25
11.1. Controller Return ResponseEntity	25

11.2. Example ResponseEntity Responses	26
11.3. Controller Exception Handler	26
11.4. Simplified Controller Using ExceptionHandler	27
12. Summary	29

Chapter 1. Introduction

You learned the meaning of web APIs and supporting concepts in the previous lecture. This module is an introductory lesson to get started implementing some of those concepts. Since this lecture is primarily implementation, I will use a set of simplistic remote procedure calls (RPC) that are **far** from REST-like and place the focus on making and mapping to HTTP calls from clients to services using Spring and Spring Boot.

1.1. Goals

The student will learn to:

- identify two primary paradigms in today's server logic: synchronous and reactive
- develop a service accessed via HTTP
- develop a client to an HTTP-based service
- access HTTP response details returned to the client
- explicitly supply HTTP response details in the service code

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. identify the difference between the Spring MVC and Spring WebFlux frameworks
2. identify the difference between synchronous and reactive approaches
3. identify reasons to choose synchronous or reactive
4. implement a service method with Spring MVC synchronous annotated controller
5. implement a client using Spring MVC RestTemplate
6. implement a client using Spring Webflux in synchronous mode
7. pass parameters between client and service over HTTP
8. return HTTP response details from service
9. access HTTP response details in client

Chapter 2. Spring Web APIs

There are two primary, overlapping frameworks within Spring for developing HTTP-based APIs:

- [Spring MVC](#)
- [Spring WebFlux](#)

Spring MVC is the legacy framework that operates using synchronous, blocking request/reply constructs. Spring WebFlux is the follow-on framework that builds on Spring MVC by adding asynchronous, non-blocking constructs that are inline with the [reactive streams paradigm](#).

2.1. Lecture/Course Focus

The focus of this lecture, module, and early portions of the course will be on synchronous communications patterns. The synchronous paradigm is simpler and there are a lot of API concepts to cover before worrying about managing the asynchronous streams of the reactive programming model. In addition to reactive concepts, Spring WebFlux brings in a heavy dose of Java 8 lambdas and functional programming that should only be applied once we master more of the API concepts.

However, we need to know the two approaches exist in order to make sense of the software and available documentation. For example, the client-side of Spring MVC (i.e., [RestTemplate](#)) has been put in "maintenance mode" (minor changes and bug fixes only) and its duties fulfilled by Spring WebFlux (i.e., [WebClient](#)). Therefore, I will be demonstrating synchronous client concepts using both libraries to help bridge the transition.



WebClient examples demonstrated here are intentionally synchronous

Examples of Spring WebFlux's [WebClient](#) will be demonstrated as a synchronous replacement for Spring MVC [RestTemplate](#). Details of the reactive API will not be covered.

2.2. Spring MVC

[Spring MVC](#) was originally implemented for writing Servlet-based applications. The term "MVC" stands for "Model, View, and Controller" — which is a standard framework pattern that separates concerns between:

- data and access to data ("the model"),
- representation of the data ("the view"), and
- decisions of what actions to perform when ("the controller").

The separation of concern provides a means to logically divide web application code along architecture boundaries. Built-in support for HTTP-based APIs have matured over time and with the shift of UI web applications to Javascript frameworks running in the browser, the focus has likely shifted towards the API development.

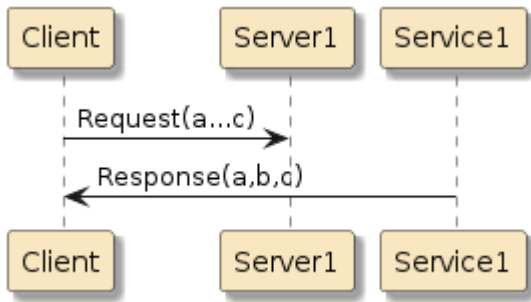


Figure 1. Spring MVC Synchronous Model

As mentioned earlier, the programming model for Spring MVC is synchronous, blocking request/reply. Each active request is blocked in its own thread while waiting for the result of the current request to complete. This mode scales primarily by adding more threads — most of which are blocked performing some sort of I/O operation.

2.3. Spring WebFlux

Spring WebFlux is built using a stream-based, reactive design as a part of Spring 5/Spring Boot 2. The reactive programming model was adopted into the [java.util.concurrent package](#) in Java 9, to go along with other asynchronous programming constructs — like `Future<T>`.

Some of the core concepts—like annotated `@RestController` and method associated annotations—still exist. The most visible changes added include the optional functional controller and the new, mandatory data input and return publisher types:

- `Mono` - a handle to a promise of a single object in the future
- `Flux` - a handle to a promise of many objects in the future

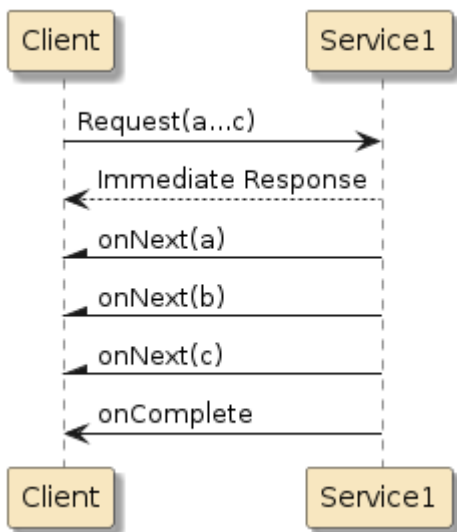


Figure 2. Spring WebFlux Reactive Model

For any single call, there is an immediate response and then a flow of events that start once the flow is activated by a subscriber. The flow of events are published to and consumed from the new mandatory `Mono` and `Flux` data input and return types. No overall request is completed using an end-to-end single thread. Work to process each event must occur in a non-blocking manner. This technique sacrifices raw throughput of a single request to achieve better performance when operating at greater concurrent scale.

2.4. Synchronous vs Asynchronous

To go a little further in contrasting the two approaches, the diagram below depicts a contrast between a call to two separate services using the synchronous versus asynchronous processing paradigms.

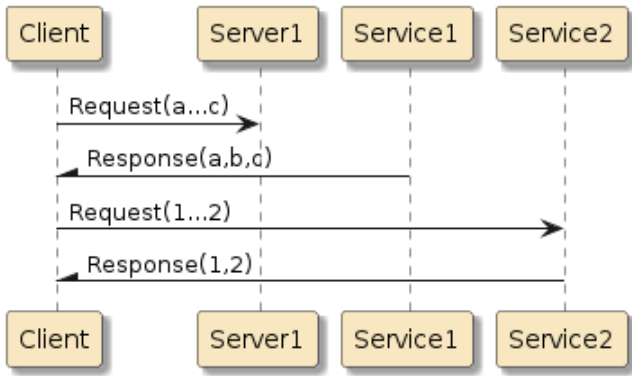


Figure 3. Synchronous

For synchronous, the call to service 2 cannot be initiated until the synchronous call/response from service 1 is completed

For asynchronous, the call to service 1 and 2 are initiated sequentially but are carried out concurrently, and completed independently

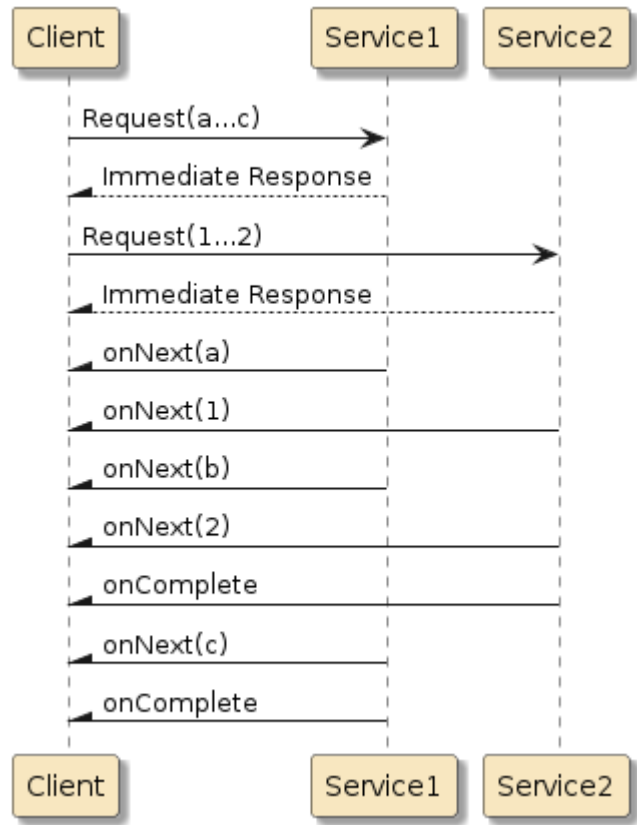


Figure 4. Asynchronous

There are different types of asynchronous processing. Spring has long supported threads with `@Async` methods. However, that style simply launches one or more additional threads that potentially also contain synchronous logic that will likely block at some point. The reactive model is strictly non-blocking—relying on the backpressure of available data and the resources being available to consume it. With the reactive programming paradigm comes strict rules of the road.

2.5. Mixing Approaches

There is a certain amount of mixture of approaches allowed with Spring MVC and Spring WebFlux. A pure reactive design without a trace of Spring MVC can operate on the [Reactor Netty](#) engine—optimized for reactive processing. Any use of Web MVC will cause the application to be considered a Web MVC application, chose between Tomcat or Jetty for the web server, and operate any use of reactive endpoints in a compatibility mode. ^[1]

With that said—functionally, we can mix Spring Web MVC and Spring WebFlux together in an application using what is considered to be the Web MVC container.

- Synchronous and reactive flows can operate side-by-side as independent paths through the code
- Synchronous flows can make use of asynchronous flows. A primary example of that is using the `WebClient` reactive methods from a Spring MVC controller-initiated flow

However, we cannot have the callback of a reactive flow make synchronous requests that can indeterminately block—or it itself will become synchronous and tie up a critical reactor thread.



Spring MVC has non-optimized, reactive compatibility

Tomcat and Jetty are Spring MVC servlet engines. Reactor Netty is a Spring WebFlux engine. Use of reactive streams within the Spring MVC container is supported—but not optimized or recommended beyond use of the `WebClient` in Spring MVC applications. Use of synchronous flows is not supported by Spring WebFlux.

2.6. Choosing Approaches

Independent synchronous and reactive flows can be formed on a case-by-case basis and optimized if implemented on separate instances. ^[1] We can choose our ultimate solution(s) based on some of the recommendations below.

Synchronous

- existing synchronous API working fine — no need to change ^[2]
- easier to learn - can use standard Java imperative programming constructs
- easier to debug - everything in same flow is commonly in same thread
- the number of concurrent users is a manageable (e.g., <100) number ^[3]
- service is CPU-intensive ^[4]
- codebase makes use of `ThreadLocal`
- service makes use of synchronous data sources (e.g., JDBC, JPA)

Reactive

- need to serve a significant number (e.g., 100-300) of concurrent users ^[3]
- requires knowledge of Java stream and functional programming APIs
- does little to no good (i.e., **badly**) if the services called are synchronous (i.e., initial response returns when overall request complete) (e.g., JDBC, JPA)
- desire to work with Kotlin or Java 8 lambdas ^[2]
- service is IO-intensive (e.g., database or external service calls) ^[4]

For many of the above reason, we will start out our HTTP-based API coverage in this course using the synchronous approach.

[1] *"Can I use SpringMvc and webflux together?"*, Brian Clozel, 2018

[2] *"Spring WebFlux Documentation - Applicability"*, version 5.2.6 release

[3] *"SpringBoot: Performance War"*, Santhosh Krishnan, 2020

[4] *"Do's and Don'ts: Avoiding First-Time Reactive Programmer Mines"*, Sergei Egorov, SpringOne Platform, 2019

Chapter 3. Maven Dependencies

Most dependencies for Spring MVC are satisfied by changing `spring-boot-starter` to `spring-boot-starter-web`. Among other things, this brings in dependencies on `spring-webmvc` and `spring-boot-starter-tomcat`.

Spring MVC Starter Dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

The dependencies for Spring MVC and Spring WebFlux's `WebClient` are satisfied by adding `spring-boot-starter-webflux`. It primarily brings in the `spring-webflux` and the reactive libraries, and `spring-boot-starter-reactor-netty`. We won't be using the netty engine, but `WebClient` does make use of some netty client libraries that are brought in when using the starter.

Spring MVC/Spring WebFlux Blend Dependency

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Chapter 4. Sample Application

To get started covering the basics of Web MVC, I am going to use a very simple, remote procedure call (RPC)-oriented, [RMM level 1](#) example where the web client simply makes a call to the service to say "hi". The example is located within the `rpc-greeter-svc` module.

```
|-- pom.xml
|-- src
|   |-- main
|   |   |-- java
|   |   |   |-- info
|   |   |   |   |-- ejava
|   |   |   |   |   |-- examples
|   |   |   |   |   |   |-- svc
|   |   |   |   |   |   |   |-- rpc
|   |   |   |   |   |   |   |   |-- GreeterApplication.java
|   |   |   |   |   |   |   |   |-- greeter
|   |   |   |   |   |   |   |   |   |-- controllers
|   |   |   |   |   |   |   |   |   |-- RpcGreeterController.java
|   |   |-- resources
|   |   |-- ...
|-- test
|   |-- java
|   |   |-- info
|   |   |   |-- ejava
|   |   |   |   |-- examples
|   |   |   |   |   |-- svc
|   |   |   |   |   |   |-- rpc
|   |   |   |   |   |   |   |-- greeter
|   |   |   |   |   |   |   |   |-- GreeterRestTemplateHttpNTest.java
|   |   |   |   |   |   |   |   |-- GreeterSyncWebClientHttpNTest.java
|   |-- resources
|   |-- ...
```

Chapter 5. Annotated Controllers

Traditional Spring MVC APIs are primarily implemented around annotated controller components. Spring has a hierarchy of annotations that help identify the role of the component class. In this case the controller class will commonly be annotated with `@RestController`, which wraps `@Controller`, which wraps `@Component`. This primarily means that the class will get automatically picked up during the component scan if it is in the application's scope.

Example Spring MVC Annotated Controller

```
package info.ejava.examples.svc.httpapi.greeter.controllers;

import org.springframework.web.bind.annotation.RestController;

@RestController
// ==> wraps @Controller
//      ==> wraps @Component
public class RpcGreeterController {
    //...
}
```

5.1. Class Mappings

Class-level mappings can be used to establish a base definition to be applied to all methods and extended by method-level annotation mappings. Knowing this, we can define the base URI path using a `@RequestMapping` annotation on the controller class and all methods of this class will either inherit or extend that URI path.

In this particular case, our class-level annotation is defining a base URL path of `/rpc/greeting`.

Example Class-level Mapping

```
...
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
@RequestMapping("rpc/greeter") ①
public class RpcGreeterController {
    ...
}
```

① `@RequestMapping.path="rpc/greeting"` at class level establishes base URI path for all hosted methods



Annotations can have alias and defaults

- `value` is an alias for `path` in the `@RequestMapping` annotation
- any time there is a **single** value expressed without a property name within an annotation, the `omitted name defaults to value`

We can use either `path`, `value`, or no name (when nothing else supplied) to express the path in `@RequestMapping`.



Annotating class can help keep from repeating common definitions

Annotations like `@RequestMapping`, applied at the class level establish a base definition for all methods of the class.

5.2. Method Request Mappings

There are two initial aspects to map to our method in our first simple example: URI and HTTP method.

Example Endpoint URI

```
GET /rpc/greeter/sayHi
```

- URI - we already defined a base URI path of `/rpc/greeter` at the class level — we now need to extend that to form the final URI of `/rpc/greeter/sayHi`
- HTTP method - this is specific to each class method — so we need to explicitly declare GET (one of the standard `RequestMethod` enums) on the class method

Example Endpoint Method Implementation

```
...  
/**  
 * This is an example of a method as simple as it gets  
 * @return hi  
 */  
@RequestMapping(path="sayHi", ①  
                method=RequestMethod.GET) ②  
public String sayHi() {  
    return "hi";  
}
```

- ① `@RequestMapping.path` at the method level appends `sayHi` to the base URI
- ② `@RequestMapping.method=GET` registers this method to accept HTTP GET calls to the URI `/rpc/greeter/sayHi`



@GetMapping is an alias for @RequestMapping(method=GET)

Spring MVC also defines a `@GetMapping` and other HTTP method-specific annotations that simply wraps `@RequestMapping` with a specific method value (e.g., `method=GET`). We can use either form at the method level.

5.3. Default Method Response Mappings

A few of the prominent response mappings can be determined automatically by the container in

simplistic cases:

response body

The response body is automatically set to the marshalled value returned by the endpoint method. In this case it is a literal String mapping.

status code

The container will return the following default status codes

- 200/OK - if we return a non-null value
- 404/NOT_FOUND - if we return a null value
- 500/INTERNAL_SERVER_ERROR - if we throw an exception

Content-Type header

The container sensibly mapped our returned String to the `text/plain` Content-Type.

Example Response Mappings Result

```
< HTTP/1.1 200 ①  
< Content-Type: text/plain;charset=UTF-8 ②  
< Content-Length: 2  
...  
hi ③
```

- ① non-null, no exception return mapped to HTTP status 200
- ② non-null java.lang.String mapped to `text/plain` content type
- ③ value returned by endpoint method

5.4. Executing Sample Endpoint

Once we start our application and enter the following in the browser, we get the expected string "hi" returned.

Example Endpoint Output

```
http://localhost:8080/rpc/greeter/sayHi  
  
hi
```

If you have access to `curl` or another HTTP test tool, you will likely see the following additional detail.

Example Endpoint HTTP Exchange

```
$ curl -v http://localhost:8080/rpc/greeter/sayHi  
...  
> GET /rpc/greeter/sayHi HTTP/1.1  
> Host: localhost:8080
```

```
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200
< Content-Type: text/plain; charset=UTF-8
< Content-Length: 2
...
hi
```

Chapter 6. RestTemplate Client

The primary point of making a callable HTTP endpoint is the ability to call that endpoint from another application. With a functional endpoint ready to go, we are ready to create a Java client and will do so within a JUnit test using Spring MVC's `RestTemplate` class in the simplest way possible.

Please note that most of these steps are true for any Java HTTP client we might use. Only the steps directly related to `RestTemplate` are specific to that topic.

6.1. JUnit Integration Test Setup

We start our example by creating an integration unit test. That means we will be using the Spring context and will do so using `@SpringBootTest` annotation with two key properties:

- `classes` - reference `@Component` and/or `@Configuration` class(es) to define which components will be in our Spring context (default is to look for `@SpringBootConfiguration`, which is wrapped by `@SpringBootApplication`).
- `webEnvironment` - to define this as a web-oriented test and whether to have a fixed (e.g., 8080), random, or none for a port number. The random port number will be injected using the `@LocalServerPort` annotation. The default value is `MOCK`—for Mock test client libraries able to bypass networking.

Example JUnit Integration Unit Test Setup

```
package info.ejava.examples.svc.rpc.greeter;

import info.ejava.examples.svc.rpc.GreeterApplication;
import lombok.extern.slf4j.Slf4j;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;

@SpringBootTest(classes = GreeterApplication.class, ①
               webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT) ②
@Tag("springboot") @Tag("greeter")
@Slf4j
public class GreeterRestTemplateHttpNTest {
    @LocalServerPort ③
    private int port;
}
```

① using the application to define the components for the Spring context

② the application will be started with a random HTTP port#

③ the random server port# will be injected into `port` annotated with `@LocalServerPort`



LocalServerPort Injection Alternatives

`@LocalServerPort` is not available until the web components are fully

initialized — which constrains how we can inject.

As you saw earlier, we can have it injected as an attribute of the test case class. This would be good if many of the `@Test` methods needed access to the raw port value.

Inject as Test Attribute

```
@SpringBootTest(...)
public class GreeterRestTemplateHttpNTest {
    @LocalServerPort
    private int port; //inject option way1
```

A close alternative would be to inject the value into the `@BeforeEach` lifecycle method. This would be good if `@Test` methods did not use the raw port value — but may use something that was built from the value.

Inject into Test Lifecycle Methods

```
@BeforeEach
public void init(@LocalServerPort int port) { //inject option way2
    baseUrl = String.format("http://localhost:%d/rpc/greeter",
port);
}
```

We could move the injection to the `@TestConfiguration`. However, since the configuration is read in before the test is initialized, we must inject it into `@Bean` factory methods (versus an attribute) and annotate the `@Bean` factory with `@Lazy`. Lazy bean factories are called on demand versus eagerly at startup.

Inject into Bean Factory using @Lazy

```
import org.springframework.context.annotation.Lazy;
...
@TestConfiguration(proxyBeanMethods = false)
public class ClientTestConfiguration {
    @Bean @Lazy
    public String baseUrl(@LocalServerPort int port) { //inject option
way3
        return String.format("http://localhost:%d/rpc/greeter", port);
    }
}
```

6.2. Form Endpoint URL

Next we will form the full URL for the target endpoint. We can take the parts we know and merge that with the injected server port number to get a full URL.


```

@LocalServerPort
private int port;

@Test
public void say_hi() {
    //given - a service available at a URL and client access
    String url = String.format("http://localhost:%d/rpc/greeter/sayHi", port); ①
    ...
}

```

① full URL to the example endpoint

6.3. Obtain RestTemplate

With a URL in hand, we are ready to make the call. We will do that using the synchronous [RestTemplate](#) from the Spring MVC library.

Spring Template is a thread safe class that can be constructed with a default constructor for the simple case—or through a [builder](#) in more complex cases and injected to take advantage of separation of concerns.

Example Obtain Simple/Default RestTemplate

```
RestTemplate restTemplate = new RestTemplate();
```

6.4. Invoke HTTP Call

There are dozens of potential calls we can make with [RestTemplate](#). We will learn many more but in this case we are

- performing an HTTP GET
- executing the HTTP method against a URL
- returning the response body content as a String

Example Invoke HTTP Call

```
String greeting = restTemplate
    .getForObject(url, String.class); ①
```

① return a String greeting from the response body of a GET URL call

Note that a successful return from [getForObject\(\)](#) will only occur if the response from the server is a 2xx/successful response. Otherwise an exception of one of the following types will be thrown:

- [RestClientException](#) - error occurred communicating with server
 - [RestClientResponseException](#) error response received from server
 - [HttpStatusCodeException](#) - HTTP response received and HTTP status known
 - [HttpServerErrorException](#) - HTTP server (5xx) errors
 - [HttpClientErrorException](#) - HTTP client (4xx) errors

- BadRequest, NotFound, UnprocessableEntity, ...

6.5. Evaluate Response

At this point we have made our request and have received our reply and can evaluate the reply against what was expected.

Evaluate Response Body

```
//then - we get a greeting response  
then(greeting).isEqualTo("hi");
```

Chapter 7. WebClient Client

The Spring 5 documentation states the `RestTemplate` is in "maintenance mode" and that we should switchover to using the Spring WebFlux `WebClient`. Representatives from Pivotal have stated in various conference talks that `RestTemplate` will likely not go away anytime soon but would likely not get upgrades to any new drivers.

In demonstrating `WebClient`, there are a few aspects of our `RestTemplate` example that do not change and I do not need to repeat.

- JUnit test setup — i.e., establishing the Spring context and random port#
- Obtaining a URL
- Evaluating returned response

The new aspects include

- obtaining the `WebClient` instance
- invoking the HTTP endpoint endpoint and obtaining result

7.1. Obtain WebClient

`WebClient` is an interface and must be constructed through a builder. A default builder can be obtained through a static method of the `WebClient` interface. `WebClient` is also thread safe, is capable of being configured in a number of ways, and its builder can be injected to create individualized instances.

Example Obtain WebClient

```
WebClient webClient = WebClient.builder().build();
```

7.2. Invoke HTTP Call

The methods for `WebClient` are arranged in a builder type pattern where each layer of call returns a type with a constrained set of methods that are appropriate for where we are in the call tree.

The example below shows an example of

- performing an HTTP GET
- targeting the HTTP methods at a specific URL
- retrieving an overall result — which is really a demarcation that the request definition is complete and from here on is the definition for what to do with the response
- retrieving the body of the result—a specification of what to do with the response when it arrives. This will be a publisher (e.g., Mono or Flux) of some sort of value or type based on the response
- blocking until the reactive response is available

Example Invoke HTTP Call

```
String greeting = webClient.get()
    .uri(url)
    .retrieve()
    .bodyToMono(String.class)
    .block(); ①
```

① Calling `block()` causes the reactive flow definition to begin producing data

The `block()` call is the synchronous part that we would look to avoid in a truly reactive thread. It is a type of subscriber that triggers the defined flow to begin producing data. This `block()` is blocking the current (synchronous) thread — just like `RestTemplate`. The portions of the call ahead of `block()` are performed in a reactive set of threads.

Chapter 8. Implementing Parameters

There are three primary ways to map an HTTP call to method input parameters:

- request body — annotated with `@RequestBody` that we will see in a POST
- path parameter — annotated with `@PathVariable`
- query parameter - annotated with `@RequestParam`

The later two are part of the next example and expressed in the URI.

Example URI with path and query parameters

```
GET /rpc/greeter/say/hello?name=jim
    / ①
    \ ②
```

① URI path segments can be mapped to input method parameters

② individual query values can be mapped to input method parameters

- we can have 0 to N path or query parameters
 - path parameters are part of the resource URI path and are commonly required when defined — but that is not a firm rule
 - query parameters are commonly the technique for optional arguments against the resource expressed in the URI path

8.1. Controller Parameter Handling

Parameters derived from the URI path require that the path be expressed with `{placeholder}` names within the string. That placeholder name will be mapped to a specific method input parameter using the `@PathVariable` annotation. In the following example, we are mapping whatever is in the position held by the `{greeting}` placeholder — to the `greeting` input variable.

Specific query parameters are mapped by their name in the URL to a specific method input parameter using the `@RequestParam` annotation. In the following example, we are mapping whatever is in the value position of `name=` to the `name` input variable.

Example Path and Query Param

```
@RequestMapping(path="say/{greeting}", ①
                 method=RequestMethod.GET)
public String sayGreeting(
    @PathVariable("greeting") String greeting, ①
    @RequestParam(value = "name", defaultValue = "you") String name) { ②
    return greeting + ", " + name;
}
```

- ① URI path placeholder `{greeting}` is being mapped to method input parameter `String greeting`
- ② URI query parameter `name` is being mapped to method input parameter `String name`



No direct relationship between placeholder/query names and method input parameter names

There is no direct correlation between the path placeholder or query parameter name and the name of the variable without the `@PathVariable` and `@RequestParam` mappings.

8.2. Client-side Parameter Handling

As mentioned above, the path and query parameters are expressed in the URL—which is not impacted whether we use `RestTemplate` or `WebClient`.

Example URL with Path and Query Params

```
http://localhost:8080/rpc/greeter/say/hello?name=jim
```

A way to build a URL through type-safe convenience methods is with the `UriComponentsBuilder` class. In the following example:

- `fromHttpUrl()` - starts the URI using a string containing the base (e.g. `http://localhost:8080/rpc/greeter`)

- `path()` - can be used to tack on a path to the end of the `baseUrl`. `replacePath()` is also a convenient method here to use when the value you have is the full path. Note the placeholder with `{greeting}` reserving a spot in the path. The position in the URI is important, but there is no direct relationship between what the client and service use for this placeholder name—if they use one at all.

- `queryParams()` - is used to express individual query parameters. The name of the query parameter must match what is expected by the service. Note that a placeholder was used here to express the value.

- `build()` - is used to finish off the URI. We pass in the placeholder values in the order they appear in the URI expression

```
@Test
public void say_greeting() {
    //given - a service available to
    //provide a greeting
    URI url = UriComponentsBuilder
        .fromHttpUrl(baseUrl)
        .path("/say/{greeting}") ①
        .queryParams("name", "{name}") ②
        .build("hello", "jim"); ③
}
```

- ① path is being expressed using a `{greeting}` placeholder for the value
- ② query parameter expressed using a `{name}` placeholder for the value
- ③ values for `greeting` and `name` are filled in during call to `build()` to complete the URI

Chapter 9. Accessing HTTP Responses

The target of an HTTP response may be a specific marshalled object or successful status. However, it is common to want to have access to more detailed information. For example:

- Success — was it a 201/CREATED or a 200/OK?
- Failure — was it a 400/BAD_REQUEST, 404/NOT_FOUND, 422/UNPROCESSABLE_ENTITY, or 500/INTERNAL_SERVER_ERROR?

Spring can supply that additional information in a `ResponseEntity<T>`, supplying us with:

- status code
- response headers
- response body — which will be unmarshalled to the specified type of `T`

To obtain that object — we need to adjust our call to the client.

9.1. Obtaining ResponseEntity

The two client libraries offer additional calls to obtain the `ResponseEntity`.

Example RestTemplate ResponseEntity<T> Call

```
//when - asking for that greeting
ResponseEntity<String> response = restTemplate.getForEntity(url, String.class);
```

Example WebClient ResponseEntity<T> Call

```
//when - asking for that greeting
ResponseEntity<String> response = webClient.get()
    .uri(url)
    .retrieve()
    .toEntity(String.class)
    .block();
```

9.2. ResponseEntity<T>

The `ResponseEntity<T>` can provide us with more detail than just the response object from the body. As you can see from the following evaluation block, the client also has access to the status code and headers.

Example Returned ResponseEntity<T>

```
//then - response be successful with expected greeting
then(response.getStatusCode()).isEqualTo(HttpStatus.OK);
then(response.getHeaders().getFirst(Headers.CONTENT_TYPE)).startsWith("text/plain");
```

```
then(response.getBody()).isEqualTo("hello, jim");
```


Chapter 10. Client Error Handling

As indicated earlier, something could fail in the call to the service and we do not get our expected response returned.

Example Response Error

```
$ curl -v http://localhost:8080/rpc/greeter/boom
...
< HTTP/1.1 400
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Thu, 21 May 2020 19:37:42 GMT
< Connection: close
<
{"timestamp":"2020-05-21T19:37:42.261+0000","status":400,"error":"Bad Request",
"message":"Required String parameter 'value' is not present" ①
...

```

① Spring MVC has default error handling that will, by default return an application/json rendering of an error

Although there are differences in their options — for the most part both `RestTemplate` and `WebClient` will throw an exception if the status code is not successful. Although very similar — unfortunately, their exceptions are technically different and would need separate exception handling logic if used together.

10.1. RestTemplate Response Exceptions

`RestTemplate` is designed to always throw an exception when there is a non-successful status code. Although we can tweak the specific exceptions thrown with filters, we are eventually forced to throw something if we cannot return an object of the requested type or a `ResponseEntity<T>` carrying the requested type.

All default `RestTemplate` exceptions thrown extend `HttpClientErrorException` — which is a `RuntimeException`, so handling the exception is not mandated by the Java language. The example below is catching a specific `BadRequest` exception (if thrown) and then handling the exception in a generic way.

Example RestTemplate Exception

```
import org.springframework.web.client.HttpClientErrorException;
...
//when - calling the service
HttpClientErrorException ex = catchThrowableOfType( ①
    ()->restTemplate.getForEntity(url, String.class),
    HttpClientErrorException.BadRequest.class);

```

① using `assertj catchThrowableOfType()` to catch the exception and it be of a specific type only if thrown



catchThrowableOfType does not fail if no exception thrown

AssertJ `catchThrowableOfType` only fails if an exception of the wrong type is thrown. It will return a null if no exception is thrown. That allows for a "BDD style" of testing where the "when" processing is separate from the "then" verifications.

10.2. WebClient Response Exceptions

`WebClient` has two primary paths to invoke a request: `retrieve()` and `exchange()`. `retrieve()` works very similar to `RestTemplate.<method>ForEntity()` — where it returns what you ask or throws an exception. `exchange()` permits some analysis of the response—but ultimately places you in a position that you need to throw an exception if you cannot return the type requested or a `ResponseEntity<T>` carrying the type requested.

All default `WebClient` exceptions thrown extend `WebClientResponseException`—which is also a `RuntimeException`, so it has that in common with the exception handling of `RestTemplate`. The example below is catching a specific `BadRequest` exception and then handling the exception in a generic way.

Example WebClient Exception

```
import org.springframework.web.reactive.function.client.WebClientResponseException;
...
//when - calling the service
WebClientResponseException.BadRequest ex = catchThrowableOfType(
    () -> webClient.get().uri(url).retrieve().toEntity(String.class).block(),
    WebClientResponseException.BadRequest.class);
```

10.3. RestTemplate and WebClient Exceptions

Once the code calling one of the two clients has the client-specific exception object, they have access to three key response values:

- HTTP status code
- HTTP response headers
- HTTP body as string or byte array

The following is an example of handling an exception thrown by `RestTemplate`.

Example RestTemplate Exception Inspection

```
HttpClientErrorException ex = ...

//then - we get a bad request
then(ex.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
```

```
then(ex.getResponseHeaders().getFirst(HttpHeaders.CONTENT_TYPE))
    .isEqualTo(MediaType.APPLICATION_JSON_VALUE);
log.info("{} ", ex.getResponseBodyAsString());
```

The following is an example of handling an exception thrown by `WebClient`.

Example WebClient Exception Inspection

```
WebClientResponseException.BadRequest ex = ...

//then - we get a bad request
then(ex.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
then(ex.getHeaders().getFirst(HttpHeaders.CONTENT_TYPE)) ①
    .isEqualTo(MediaType.APPLICATION_JSON_VALUE);
log.info("{} ", ex.getResponseBodyAsString());
```

① `WebClient` 's exception method name to retrieve response headers different from `RestTemplate`

Chapter 11. Controller Responses

In our earlier example, our only response option from the service was a limited set of status codes derived by the container based on what was returned. The specific error demonstrated was generated by the Spring MVC container based on our mapping definition. It will be common for the controller method, itself to need explicit control over the HTTP response returned --primarily to express response-specific

- HTTP status code
- HTTP headers

11.1. Controller Return ResponseEntity

The following service example performs some trivial error checking and:

- responds with an explicit error if there is a problem with the input
- responds with an explicit status and Content-Location header if successful

The service provides control over the entire response by returning a `ResponseEntity` containing the complete HTTP result versus just returning the result value for the body. The `ResponseEntity` can express status code, headers, and the returned entity.

Example Controller Returning ResponseEntity

```
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
...
@RequestMapping(path="boys",
    method=RequestMethod.GET)
public ResponseEntity<String> createBoy(@RequestParam("name") String name) { ①
    try {
        someMethodThatMayThrowException(name);

        String url = ServletUriComponentsBuilder.fromCurrentRequest() ②
            .build().toUriString();
        return ResponseEntity.ok() ③
            .header(HttpHeaders.CONTENT_LOCATION, url)
            .body(String.format("hello %s, how do you do?", name));
    } catch (IllegalArgumentException ex) {
        return ResponseEntity.unprocessableEntity() ④
            .body(ex.toString());
    }
}
private void someMethodThatMayThrowException(String name) {
    if ("blue".equalsIgnoreCase(name)) {
        throw new IllegalArgumentException("boy named blue?");
    }
}
```

- ① `ResponseEntity` returned used to express full HTTP response
- ② `ServletUriComponentsBuilder` is a URI builder that can provide context of current call
- ③ service is able to return an explicit HTTP response with appropriate success details
- ④ service is able to return an explicit HTTP response with appropriate error details

11.2. Example `ResponseEntity` Responses

In the response we see the explicitly assigned status code and Content-Location header.

Example `ResponseEntity` Success Returned

```
curl -v http://localhost:8080/rpc/greeter/boys?name=jim
...
< HTTP/1.1 200 ①
< Content-Location: http://localhost:8080/rpc/greeter/boys?name=jim ②
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 25
...
hello jim, how do you do?
```

- ① status explicitly
- ② Content-Location header explicitly supplied by service

For the error condition, we see the explicit status code and error payload assigned.

Example `ResponseEntity` Error Returned

```
$ curl -v http://localhost:8080/rpc/greeter/boys?name=blue
...
< HTTP/1.1 422 ①
< Content-Type: text/plain;charset=UTF-8
< Content-Length: 15
...
boy named blue?
```

- ① HTTP status code explicitly supplied by service

11.3. Controller Exception Handler

We can make a small but substantial step at simplifying the controller method by making sure the exception thrown is fully descriptive and moving the exception handling to either a separate, annotated method of the controller or globally to be used by all controllers (shown later).

The following example uses `@ExceptionHandler` annotation to register a handler for when controller methods happen to throw the `IllegalArgumentException`. The handler has the ability to return an explicit `ResponseEntity` with the error details.

Example Controller ExceptionHandler

```
import org.springframework.web.bind.annotation.ExceptionHandler;
...
ExceptionHandler(IllegalArgumentException.class) ①
public ResponseEntity<String> handle(IllegalArgumentException ex) {
    return ResponseEntity.unprocessableEntity() ②
        .body(ex.getMessage());
}
```

① `ExceptionHandler` is registered to handle all `IllegalArgumentException` exceptions thrown by controller method (or anything it calls)

② handler builds a `ResponseEntity` with the details of the error



Create custom exceptions to address specific errors

Create custom exceptions to the point that the handler has the information and context it needs to return a valuable response.

11.4. Simplified Controller Using ExceptionHandler

With all exceptions addressed by `ExceptionHandler`s, we can free our controller methods of tedious, repetitive conditional error reporting logic and still return an explicit HTTP response.

Example Controller Method using ExceptionHandler

```
@RequestMapping(path="boys/throws",
    method=RequestMethod.GET)
public ResponseEntity<String> createBoyThrows(@RequestParam("name") String name) {
    someMethodThatMayThrowException(name); ①

    String url = ServletUriComponentsBuilder.fromCurrentRequest()
        .replacePath("/rpc/greeter/boys") ②
        .build().toUriString();

    return ResponseEntity.ok()
        .header(HttpHeaders.CONTENT_LOCATION, url)
        .body(String.format("hello %s, how do you do?", name));
}
```

① Controller method is free from dealing with exception logic

② replacing path in order to match sibling implementation response

Note the new method endpoint with the exception handler returns the same, explicit HTTP response as the earlier example.

Example ExceptionHandler Response

```
curl -v http://localhost:8080/rpc/greeter/boys/throws?name=blue
```

```
...  
< HTTP/1.1 422  
< Content-Type: text/plain;charset=UTF-8  
< Content-Length: 15  
...  
boy named blue?
```

Chapter 12. Summary

In this module we:

- identified two primary paradigms (synchronous and reactive) and web frameworks (Spring MVC and Spring WebFlux) for implementing web processing and communication
- implemented an HTTP endpoint for a URI and method using Spring MVC annotated controller in a fully synchronous mode
- demonstrated how to pass parameters between client and service using path and query parameters
- demonstrated how to pass return results from service to client using http status code, response headers, and response body
- demonstrated how to explicitly set HTTP responses in the service
- demonstrated how to clean up service logic by using exception handlers
- demonstrated how to invoke methods from a Spring MVC `RestTemplate` and Spring WebFlux `WebClient`