Spring MVC

jim stafford

Fall 2024 v2022-07-15: Built: 2024-11-19 21:31 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 14
   6.3. Obtain RestTemplate 14
   6.4. Invoke HTTP Call 15
   6.5. Evaluate Response 15
7. Spring Rest Clients 16
8. RestClient Client 17
   8.1. Obtain RestClient 17
   8.2. Invoke HTTP Call 17
9. WebClient Client 19
   9.1. Obtain WebClient 19
   9.2. Invoke HTTP Call 19
10. Spring HTTP Interface 21
11. Implementing Parameters 22
   11.1. Controller Parameter Handling 22
   11.2. Client-side Parameter Handling 23
   11.3. Spring HTTP Interface Parameter Handling 24
12. Accessing HTTP Responses 25
   12.1. Obtaining ResponseEntity 25
   12.2. ResponseEntity<T> 26

13. Client Error Handling . . . 27
13.1. RestTemplate Response Exceptions . . . 27
13.2. RestClient Response Exceptions . . . 28
13.3. WebClient Response Exceptions . . . 29
13.4. Spring HTTP Interface Exceptions . . . 30
13.5. Client Exceptions . . . 30
14. Controller Responses . . . 32
14.1. Controller Return ResponseEntity . . . 32
14.2. Example ResponseEntity Responses . . . 33
14.3. Controller Exception Handler . . . 33
14.4. Simplified Controller Using ExceptionHandler . . . 34
15. Summary . . . 36

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 synchronous client using RestTemplate API
6. implement a synchronous client using RestClient fluent API
7. implement a client using Spring Webflux fluent API in synchronous mode
8. pass parameters between client and service over HTTP
9. return HTTP response details from service
10. access HTTP response details in client
11. implement exception handler outside of service method

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 most portions of the course will be on synchronous communications patterns. The synchronous paradigm is simpler, and there are a ton 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 to make sense of the software and available documentation. For example, the long-time legacy client-side of Spring MVC (i.e., RestTemplate) was put in "maintenance mode" (minor changes and bug fixes only) towards the end of Spring 5, with its duties fulfilled by Spring WebFlux (i.e., WebClient). Spring 6 introduced a middle ground with `RestClient` that addresses the synchronous communication simplicity of `RestTemplate` with the fluent API concepts of `WebClient`.

It is certain that you will encounter use of `RestTemplate` in legacy Spring applications and there is no strong reason to replace. There is a good chance you may have the desire to work with a fluent or reactive API. Therefore, I will be demonstrating synchronous client concepts using each library to help cover all bases.

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 has 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.

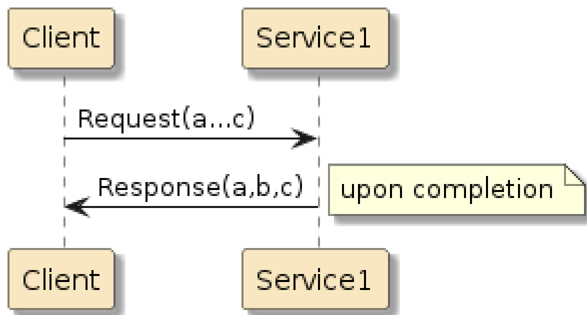

Figure 1. Spring MVC Synchronous Model

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

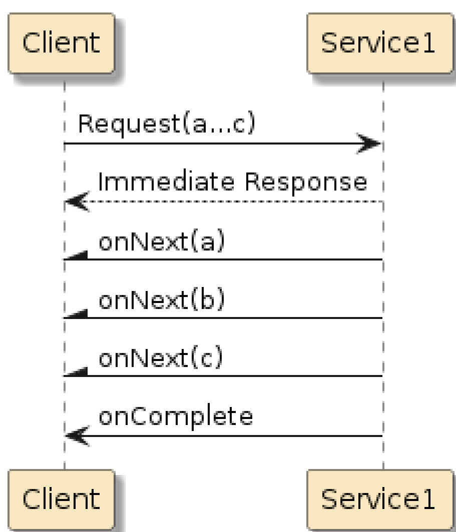

Figure 2. Spring WebFlux Reactive Model

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 is 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 a 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.

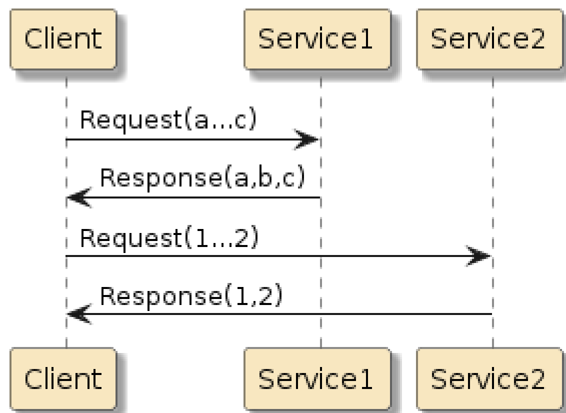

Figure 3. Synchronous

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

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

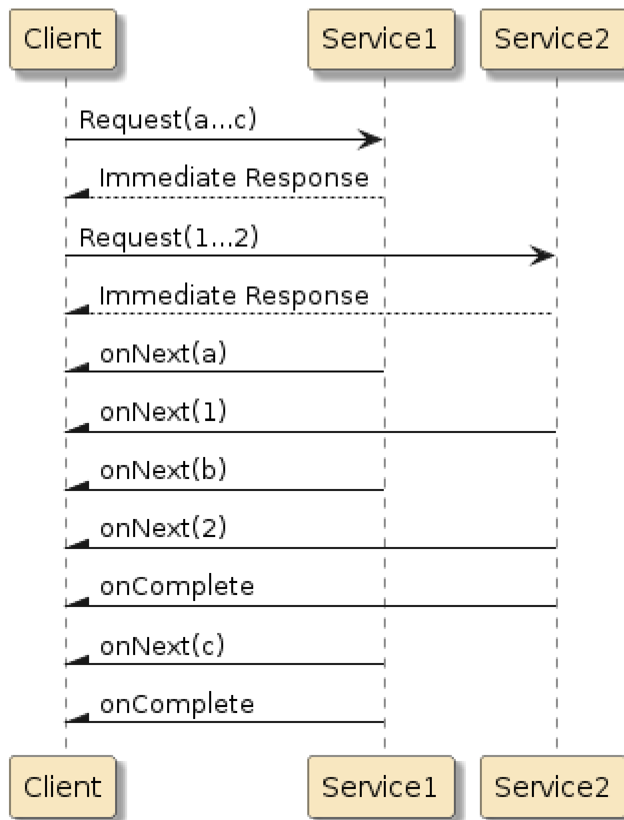

Figure 4. Asynchronous

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, choose 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 programing constructs
- easier to debug - everything in the same flow is commonly in the 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 reasons, 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 basic, 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 ①
    |   |                               `-- RpcGreeterController.java
    |   `-- resources
    |       `-- ...
    `-- test
        |-- java
        |   `-- info
        |       `-- ejava
        |           `-- examples
        |               `-- svc
        |                   `-- rpc
        |                       `-- greeter ②
        |                           |-- GreeterRestTemplateNTest.java
        |                           |-- GreeterRestClientNTest.java
        |                           |-- GreeterSyncWebClientNTest.java

        |                           |-- GreeterHttpIfaceNTest.java
        |                           |-- GreeterAPI.java

        |                           `-- ClientTestConfiguration.java
        `-- resources
                `-- ...
```

① example @RestController

② example clients using RestTemplate, RestClient, WebClient, and Http Interface Proxy

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 path for all HTTP-accessible 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. I will go through all the steps for `RestTemplate` here but only cover the unique aspects to the alternate techniques later on.

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 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
public class GreeterRestTemplateNTest {
    @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 is alias for Property local.server.port

`@LocalServerPort` annotation acts as an alias for the `local.server.port` property.

```
package org.springframework.boot.test.web.server;
import org.springframework.beans.factory.annotation.Value;
```

```
...
@Value("${local.server.port}")
public @interface LocalServerPort {}
```

One could use that property instead to express the injection.

```
@Value("${local.server.port}")
private int port;
```

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 GreeterRestTemplateNTest {
    @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.

Create @Bean Factory using @LocalServerPort and @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);
    }
```

Inject @Bean into Test Case

```
@SpringBootTest(classes = {GreeterApplication.class, //optionally
naming app config
        ClientTestConfiguration.class},
        webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class GreeterRestTemplateNTest {
    @Autowired @Qualifier("baseUrl") //qualifier makes bean selection
more explicit
    private String injectedBaseUrl; //initialized in test config using
way3
```

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.

Forming endpoint URL with String.format()

```
@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

Starting Simple

Starting simple. We will be covering more type-safe, purpose-driven ways to perform related client actions in this and follow-on lectures.

6.3. Obtain RestTemplate

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

Spring's `RestTemplate` 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

```
import org.springframework.web.client.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

6.4.1. Exceptions

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 occured 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. Spring Rest Clients

The Spring 5 documentation stated `RestTemplate` was going into "maintenance mode" and that we should switchover to using the Spring WebFlux `WebClient`. The current Spring 6 documentation dropped that guidance and made the choice driven by:

- synchronous - RestTemplate
- fluent and synchronous - RestClient, new in Spring 6.1
- fluent and asynchronous/reactive - WebClient

Spring 6 also added features to all three for:

- client-side API facade - HTTP Interface - provides a type-safe business interface to any of the clients

I will summarize these additions next.

Chapter 8. RestClient Client

RestClient is a synchronous API like `RestTemplate`, but works using fluent ("chaining"; *client.m1().m2()*) API calls like `WebClient`. The asynchronous `WebClient` fluent API was introduced in Spring 5 and `RestClient` followed in Spring 6.1. When using `WebClient` in synchronous mode — the primary difference with `RestClient` is no need to explicitly block for exchanges to complete.

In demonstrating `RestClient`, 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 `RestClient` instance
- invoking the HTTP endpoint and obtaining result

8.1. Obtain RestClient

`RestClient` is an interface and must be constructed through a builder. A default builder can be obtained through a static method of the `RestClient` interface. `RestClient` 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 RestClient

```
import org.springframework.web.client.RestClient;
...
    RestClient restClient = RestClient.builder().build();
```

If you are already invested in a detailed `RestTemplate` setup of configured defaults and want the fluent API, `RestClient` can be constructed from an existing `RestTemplate` instance.

Example Building RestClient from RestTemplate

```
RestTemplate restTemplate = ...
RestClient restClient=RestClient.create(restTemplate);
```

8.2. Invoke HTTP Call

The methods for `RestClient` 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 the type to expect

Example Invoke HTTP Call

```
String greeting = restClient.get()
        .uri(url)
        .retrieve()
        .body(String.class);
```

Chapter 9. WebClient Client

`WebClient` and `RestClient` look and act very much the same, with the primary difference being the reactive/asynchronous API aspects for `WebClient`.

9.1. 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

```
import org.springframework.web.reactive.function.client.WebClient;
...
    WebClient webClient = WebClient.builder().build();
```

One cannot use a `RestTemplate` or `RestClient` instance to create a `WebClient`. They are totally different threading models under the hood.

9.2. Invoke HTTP Call

The fluent API methods for `WebClient` are much the same as `RestClient` except for when it comes to obtaining the payload body.

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()
//same
        .uri(url)
//same
        .retrieve()
//same
        .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 10. Spring HTTP Interface

This last feature (HTTP Interface) allows you to define a typed interface for your client API using a Java interface, annotations, and any of the Spring client APIs we have just discussed. Spring will implement the details using dynamic proxies (discussed in detail much later in the course).

We can define a simple example using our `/sayHi` endpoint by defining a method with the information required to make the HTTP call. This is very similar to what is defined on the server-side.

Example Type-safe Client Interface

```
import org.springframework.web.service.annotation.GetExchange;

interface MyGreeter {
    @GetExchange("/sayHi")
    String sayHi();
};
```

We then build a `RestTemplate`, `RestClient`, or `WebClient` by any means and assign it a baseUrl. The baseUrl plus `@GetExchange` value must equal the server-side URL.

Build Client with Base URL

```
String url = ...
RestClient restClient = RestClient.builder().baseUrl(url).build();
```

We then can create an instance of the interface using the lower-level API, `RestClientAdapter`, and `HttpServiceProxyFactory`.

Create Proxy

```
import org.springframework.web.client.support.RestClientAdapter;
import org.springframework.web.service.invoker.HttpServiceProxyFactory;
...
RestClientAdapter adapter = RestClientAdapter.create(restClient);
HttpServiceProxyFactory factory = HttpServiceProxyFactory.builderFor(adapter).build();
MyGreeter greeterAPI = factory.createClient(MyGreeter.class);
```

At this point we can call it like any Java instance/method.

Example Call to Spring HTTP Interface

```
//when - calling the service
String greeting = greeterAPI.sayHi();
```

The Spring HTTP Interface is extremely RPC-oriented, but we can make it REST-like enough to be useful. Later examples in this lecture will show some extensions.

Chapter 11. 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

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

11.1. 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. Having them match makes the mental mapping easier, but the value for the internet client URI name may not be the best value for the internal Java controller variable name.

11.2. 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`, `RestClient`, 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.
- queryParam() - 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

Example Client Code Forming URL with Path and Query Params

```
@Test
public void say_greeting() {
    //given - a service available to provide a greeting
    URI url = UriComponentsBuilder.fromHttpUrl(baseUrl)
            .path("/say/{greeting}") ①
            .queryParam("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

11.3. Spring HTTP Interface Parameter Handling

We can address parameters in Spring HTTP Interface using the same `@PathVariable` and `RequestParam` declarations that were used on the server-side. The following example shows making each of the parameters required. Notice also that we can have the call return the ResponseEntity wrapper versus just the value.

Using Required Client-side Parameter Values

```
@GetExchange("/say/{greeting}")
ResponseEntity<String> sayGreeting(
        @PathVariable(value = "greeting", required = true) String greeting,
        @RequestParam(value = "name", required=true) String name);
```

With the method defined, we can call it like a normal Java method and inspect the response.

```
//when - asking for that greeting with required parameters
... = greeterAPI.sayGreeting("hello","jim");
//response "hello, jim"
```

11.3.1. Optional Parameters

We can make parameters optional, allowing the client to null them out. The following example shows the client passing in a null for the name — to have it defaulted by either the client or server-side code.

Optional Query Parameter Call

```
//when - asking for that greeting using client-side or server-side defaults
... = greeterAPI.sayGreeting("hello", null);
```

The optional parameter can be resolved:

- on the server-side. In this case, the client marks the parameter as not required.

  Using Server-side Default Parameter Value

  ```
  @RequestParam(value = "name", required=false) String name);
  //response "hello, you"
  ```

- on the client-side. In this case, the client identifies the default value to use.

  Using Client-side default

  ```
  @RequestParam(value = "name", defaultValue="client") String name);
  //response "hello, client"
  ```

Chapter 12. 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.

12.1. Obtaining ResponseEntity

The 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 RestClient ResponseEntity<T> Call

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

Example WebClient ResponseEntity<T> Call

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

Example Spring HTTP Interface Call

```
//when - asking for that greeting
```

```
ResponseEntity<String> response = greeterAPI.sayGreeting("hello","jim");
```

12.2. 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(HttpHeaders.CONTENT_TYPE)).startsWith("text/plain"
);
then(response.getBody()).isEqualTo("hello, jim");
```

Chapter 13. 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 — `RestTemplate`, `RestClient`, and `WebClient` will throw an exception if the status code is not successful. Although very similar — unfortunately, `WebClient` exceptions are technically different than the others and would need separate exception handling logic if used together.

13.1. RestTemplate Response Exceptions

`RestTemplate` and `RestClient` will throw an exception, by default for error responses.

13.1.1. Default RestTemplate Exceptions

All non-`WebClient` 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);

//when - calling the service
HttpClientErrorException ex = catchThrowableOfType(
```

```
        () -> restClient.get().uri(url).retrieve().toEntity(String.class),
        HttpClientErrorException.BadRequest.class);
```

① using assertj `catchThrowableOfType()` to catch the exception and test that 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.

13.1.2. Noop RestTemplate Exceptions

`RestTemplate` is the only client option that allows one to bypass the exception rule and obtain an error ResponseEntity from the call without exception handling. The following example shows a `NoOpResponseErrorHandler` error handler being put in place and the caller is receiving the error `ResponseEntity` without using exception handling.

Example Bypass Exceptions

```
//configure RestTemplate to return error responses, not exceptions
RestTemplate noExceptionRestTemplate = new RestTemplate();
noExceptionRestTemplate.setErrorHandler(new NoOpResponseErrorHandler());

//when - calling the service
Assertions.assertDoesNotThrow(()->{
    ResponseEntity<String> response = noExceptionRestTemplate.getForEntity(url,
String.class);
    //then - we get a bad request
    then(response.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
    then(response.getHeaders().getFirst(HttpHeaders.CONTENT_TYPE))
        .isEqualTo(MediaType.APPLICATION_JSON_VALUE);
},"return response, not exception");
```

13.2. RestClient Response Exceptions

`RestClient` has two primary paths to invoke a request: `retrieve()` and `exchange()`.

13.2.1. RestClient retrieve() and Exceptions

`retrieve().toEntity(T)` works very similar to `RestTemplate.<method>ForEntity()` — where it returns what you ask or throws an exception. The following shows a case where the `RestClient` call will be receiving an error and throwing a `BadRequest` exception.

Default RestClient Exception Behavior with retrieve().toEntity()

```
HttpClientErrorException ex = catchThrowableOfType(
        () -> restClient.get().uri(url).retrieve().toEntity(String.class),
```

```
        HttpClientErrorException.BadRequest.class);
```

13.2.2. RestClient exchange() method

`exchange()` permits some analysis and handling of the response within the pipeline. However, it ultimately places you in a position that you need to throw an exception if you cannot return the type requested or a `ResponseEntity`. The following example shows an error being handled without an exception. One must be careful doing this since the error response likely will not be the data type requested in a realistic scenario.

Example Use of exchange() to bypass Exceptions

```
ResponseEntity<?> response = restClient.get().uri(url)
    .exchange((req, resp) -> {
        return ResponseEntity.status(resp.getStatusCode())
                .headers(resp.getHeaders())
                .body(StreamUtils.copyToString(resp.getBody(), Charset.defaultCharset
()));
    });
then(ex.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
```

All default `RestClient` exceptions thrown are identical to `RestTemplate` exceptions.

13.3. WebClient Response Exceptions

`WebClient` has the same 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. Overriding the exception handling design of these clients is not something I would recommend, and overriding the async API of the WebClient can be daunting. Therefore, I am just going to show the exception handling option.

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);
```

All default `WebClient` exceptions extend WebClientResponseException — which is also a `RuntimeException`, so it has that in common with the exception handling of `RestTemplate`.

13.4. Spring HTTP Interface Exceptions

The Spring HTTP Interface API exceptions will be identical to `RestTemplate` and `RestClient`. Any special handling of error responses can be done in the client error handling stack (e.g., RestClient.defaultStatusHandler). That will provide a means to translate the HTTP error response into a business exception if desired.

Example Spring HTTP Interface Exception

```
//when - calling the service
RestClientResponseException ex = catchThrowableOfType(
        () -> greeterAPI.boom(),
        HttpClientErrorException.BadRequest.class);
```

13.5. Client 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` and `RestClient`.

Example RestTemplate/RestClient 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 14. 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

14.1. 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

14.2. Example ResponseEntity Responses

In 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

14.3. Controller Exception Handler

We can make a small but significant 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 can 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) {②
        return ResponseEntity.unprocessableEntity() ③
                .body(ex.getMessage());
    }
```

① handles all `IllegalArgumentEception`-s thrown by controller method (or anything it calls)

② input parameter is concrete type or parent type of handled exception

③ 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.

14.4. Simplified Controller Using ExceptionHandler

With all exceptions addressed by `ExceptionHandlers`, 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 a path 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 15. 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 use of the synchronous Spring MVC `RestTemplate` and `RestClient` and reactive Spring WebFlux `WebClient` client APIs
- demonstrated use of Spring HTTP Interface to wrap low-level client APIs with a type-safe, business interface