Spring Framework REST API Development

For single page web applications or any consumers

A Complete Blueprint

Essential code and patterns for developing elegant REST APIs for your JavaScript web applications or any consumers, using the best practices of Spring Framework and Spring Boot 1.5

Second Edition, Fourth Revision (Spring Boot 1.5, Spring Lemon 0.11) - March 2017

Spring Framework for the Real World – Beginner to Expert, Module III

Sanjay Patel

www.naturalprogrammer.com

Contents

CONSIDERING SPRING FOR YOUR NEXT PROJECT?

We can provide all kinds of help you may need, starting from individual mentoring to full solutions. Click here for more details, and feel free to contact us by mailing to info@naturalpogrammer.com.

1 Introduction

Why this book

A modern JavaScript web application or a mobile app would need a robust backend API. For developing such an API, Spring Framework would be a great choice today.

But, the default configurations of Spring favor developing traditional applications, with server-side templating, like JSP or Thymeleaf[1]. When developing APIs instead, you will need to override these defaults by doing some subtle configurations. You also will need to layout various patterns for handling validation, security, etc. in an elegant manner. If you are planning to develop more than one application, you also would like to separate out this common code, as well as other common code, e.g. the user module, into a separate library.

To relieve you of all this non-functional job, we thought to bring out an open source library, named Spring Lemon, holding all these essential code and patterns. It aims to use the latest best practices found in the official guides, books, blog posts, etc., blended with our experience. It also includes a production grade user module with features like sign up, sign in, verify email, update profile, forgot password, change password, change email, captcha validation etc..

So, you can just include it in your project and jump to coding your business logic straightaway! It is quite configurable and extensible, suiting most common projects.

In this book, we will discuss bit-by-bit how Spring Lemon is developed.

This book also serves as the Module III of our Spring Framework for the Real World – Beginner to Expert course. By going through this book, you will

1. Gain the essential knowledge of developing elegant REST APIs using the latest best practices of Spring Framework and Spring Boot.
2. Learn how to develop a production grade user module.
3. Learn how to effectively put the common code into a separate library.
4. Understand *Spring Lemon* in depth, so that you can
    a. use it fluently in your applications, with all the advanced configuration and customization techniques
    b. fork and change it to your needs
    c. contribute to it

[1] For example, the *CSRF* token is attached as a request parameter, and the `<form:form>` JSP tag embeds it to a form automatically. So, when developing a JSP application, it simply works.

Prerequisites

Prior Knowledge of Spring Framework

To follow this book, you must have some prior knowledge of Spring Framework's Dependency Injection, Java configuration, Boot, MVC, Security and Spring Data JPA. If you are new to any of these, you should first take the Module I and II of our Spring Framework for the Real World – Beginner to Expert course.

Getting Started With Spring Lemon

Before proceeding, it will be highly beneficial to go through Spring Lemon's getting started guide.

Help improve Spring Lemon

Spring Lemon is a continuous endeavor – it would keep improving along with new Spring releases. We will also keep it updating as we discover better practices through our experience, new guides, books, tutorials, etc., and of course, through your feedback. If you see anything that could be improved in Spring Lemon or in this book, please let us know!

What this book is not

The main focus of this book is Spring, and not REST. Here we won't discuss the REST architectural patterns, or attempt to be 100% RESTful. We are not going to cover topics like REST semantics, API versioning or API documentation. We won't cover stateless authentication either.

> "*But, shouldn't my APIs be 100% RESTful?*" - You may ask.
>
> Not necessarily. If fact, many people advocate against trying to be stateless or 100% RESTful. Here are some interesting references, from the Spring core team:
>
> It very definitely is a Good Thing to use the session for authentication and CSRF protection
>
> The State of Securing RESTful APIs with Spring

Nonetheless, whether you want to be 100% RESTful or not, this book would serve as your first essential step.

2 How to create a library

> If you are planning just to use Spring Lemon, or fork and customize it, you may not need to create a similar new library from scratch. But, in case you sometime need to, this chapter could help.

If you have more than one Spring Boot applications, they may be having many functionalities in common – e.g. the user module. Rather than replicating that common code, keeping that as a separate library would make sense. Spring Lemon is such a library.

Such a library project would be nothing but a simple maven or gradle project with a JAR packaging. It can be created using *maven-archetype-quickstart*. If you are using *Spring Tool Suite* (*STS*), it would be easier to generate the project by using its *Spring Starter Project Wizard*, and then do some modifications to make it a library. Below are the precise steps that we had followed when creating Spring Lemon.

Creating a new Spring Boot project

We first created a new project by using the *New Spring Starter Project* wizard, available through *File -> New -> Spring Starter Project*, and filling in the following information:

Name	`spring-lemon`
Type	`Maven Project`
Packaging	`Jar`
Java Version	`1.8`
Group	`com.naturalprogrammer.spring`
Artifact	`spring-lemon`
Description	A library for Spring applications
Package	`com.naturalprogrammer.spring.lemon`
Boot Version	`1.5.1`
Dependencies	Core - Security, Data – JPA, , I/O – Mail, Web - Web

Making it a library

The following steps were then taken to convert the generated application into a library.

Removing extra code

In *pom.xml*, `spring-boot-maven-plugin`, the `pluginRepositories` section if any, and the `spring-boot-starter-test` dependency were removed, In other words, things struck out below were removed:

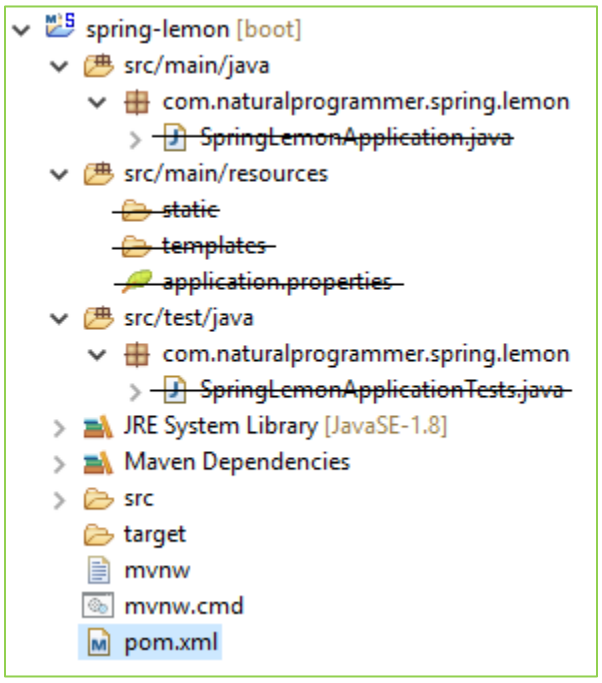
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>

      <groupId>com.naturalprogrammer.spring</groupId>
      <artifactId>spring-lemon</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <packaging>jar</packaging>

      <name>spring-lemon</name>
      <description>A library for Spring applications</description>

      <parent>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>1.5.1.RELEASE</version>
            <relativePath/> <!-- lookup parent from repository -->
      </parent>

      <properties>
            <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
            <java.version>1.8</java.version>
      </properties>

      <dependencies>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-data-jpa</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-mail</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-security</artifactId>
            </dependency>
            <dependency>
                  <groupId>org.springframework.boot</groupId>
                  <artifactId>spring-boot-starter-web</artifactId>
            </dependency>

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

```
		</dependency>
	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.springframework.boot</groupId>
				<artifactId>spring-boot-maven-plugin</artifactId>
			</plugin>
		</plugins>
	</build>

</project>
```

`spring-boot-maven-plugin` creates an executable jar, which we won't need. Also, we didn't need the `spring-boot-starter-test` dependency, because we thought not to code any test cases in Spring Lemon, and instead code integration tests in a sample application using the library.

See how Spring Lemon's pom.xml looks without these.

Also the following struck out files were removed:



Specifically, the removed files were

1. The one having the main method for running the application, (i.e. `SpringLemonApplication.java`).
2. `static` and `templates` folders if there, and `application.properties` inside `src/main/resources`. These had to be there in the application, not here in the library.
3. The generated test file (i.e. `SpringLemonApplicationTests.java`).

Compiling with *parameters* flag

We thought to configure the `maven-compiler-plugin` for compiling our code with the Java 8 `parameters` flag, so that the method parameter names would be preserved in the JAR. So, the following plugin was added to *pom.xml*:

```
...

<build>
    <plugins>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <compilerArgs>
                    <arg>-parameters</arg>
                </compilerArgs>
            </configuration>
        </plugin>

    </plugins>
</build>

...
```

Spring would then be able to use the parameter names, enabling us to shorten our code at some places. For example,

```
@PostMapping("/users/{verificationCode}/verify")
public U verifyUser(@PathVariable("verificationCode") String verificationCode) {
```

can now be shortened as

```
@PostMapping("/users/{verificationCode}/verify")
public U verifyUser(@PathVariable String verificationCode) {
```

Have a look at the final pom.xml of Spring Lemon It also includes some dependencies like `joda-time`, which we will talk about later.

Adding the library to other projects

Spring Lemon Getting Started guide shows how to add the library to an application.

3 Some Utility Methods

The LemonUtil class of Spring Lemon provides some useful utility methods. In this chapter, we are going to discuss those.

getBean

We can obtain the reference to a Spring bean in our code just by injecting it, e.g. by using *@Autowired*. But, this works only inside component classes – i.e. in other bean classes. However, sometimes we may need to use a bean in a normal class that isn't going to be a component, e.g. a POJO entity class.

So, it would be helpful to have a utility method to get a bean *dynamically*. A static `getBean` method is available in Spring Lemon for this, for getting a bean *by type*:

```
@Component
public class LemonUtil {

	private static ApplicationContext applicationContext;

	@Autowired
	public LemonUtil(ApplicationContext applicationContext) {

		LemonUtil.applicationContext = applicationContext;
	}

	public static <T> T getBean(Class<T> clazz) {
		return applicationContext.getBean(clazz);
	}
}
```

Notice that

1. Our *getBean* method uses the getBean method of `ApplicationContext`, which fetches a bean by type. If you have more than one bean of same type, you might like to fetch a bean by name instead. For this, you can code another *getBean*, and use therein the corresponding getBean method of ApplicationContext.
2. The ApplicationContext has been injected as a *static* variable, which lets us define our `getBean` method as static.

Below is an example how you can use our *getBean* method:

```
LemonService lemonService = LemonUtil.getBean(LemonService.class);
```

> *Caution*: Accessing the application context manually, as we have done above, is considered a bad practice. So, use our *getBean* method sparingly.

afterCommit

Typically, we will be making our service methods transactional, and sometimes we would want to execute come code only if the transaction succeeds. Let's see a *user signup* case:

1. Our API receives the signup request in a controller method
2. The controller method calls a service method for doing the job
3. A transaction begins
4. The service method
    a. saves the new user to the database
    b. sends a verification mail to the user
    c. exits
5. The transaction tries to commit, FAILS for some reason, and rolled back!
6. The controller receives the exception and passes it to the client
7. User sees that he was unable to sign up

We should execute *4.b*, i.e. sending the verification mail, only if the transaction succeeds. Otherwise the user will get a verification mail *even if signup fails*.

To tell Spring to run such code only after the transaction succeeds, we should put the code in the `afterCommit` method of a `TransactionSynchronizationAdapter` object, and register the object. Precisely, we should code *4.b* like this:

```
TransactionSynchronizationManager.registerSynchronization(
    new TransactionSynchronizationAdapter() {
        @Override
        public void afterCommit() {
            // code for sending verification mail
        }
});
```

This is heavy boilerplate code. Using Java 8, Spring Lemon encapsulates it into a utility method, as below:

```
public static void afterCommit(Runnable runnable) {
      TransactionSynchronizationManager.registerSynchronization(
          new TransactionSynchronizationAdapter() {
              @Override
              public void afterCommit() {
                  runnable.run();
              }
      });
}
```

`afterCommit` can then be used as below:

```
LemonUtil.afterCommit(() -> {
   // code for sending verification mail
});
```

getMessage

We assume that you already know the basics of *I18n*. To support *I18n* in a Spring Boot application, you would keep your messages as properties in *messages.properties* files, inside the `src/main/resources` folder by default. You will also have a *messages_locale.properties* for each *locale* you want to support.

messages.properties would look as below:

```
alreadyVerified: Already verified
wrong.verificationCode: Wrong verification code

userNotFound: User not found
invalidSize: Size must be between {0} and {1} characters
```

Each entry above has a *key* and a *message*. Messages can contain *placeholders*, as you see `{0}` and `{1}` above in the last line.

In our Java code, we should be able to fetch the messages by using the keys. Spring Boot autoconfigures a MessageSource, which can be used for that:

```
@Autowired
private MessageSource messageSource;

messageSource.getMessage(messageKey, args, LocaleContextHolder.getLocale());
```

The first parameter of `getMessage` is the key, the second one is an array of arguments for any placeholders in the message, and the third one is the locale of the current-user.

This would be used often, and so we can encapsulate the above code in a simpler utility method. Spring Lemon has a static `getMessage` utility mehod for this, looking as below:

```
@Component
public class LemonUtil {

	private static MessageSource messageSource;

	@Autowired
	public LemonUtil(MessageSource messageSource) {
		LemonUtil.messageSource = messageSource;
	}

	public static String getMessage(String messageKey, Object... args) {

		return messageSource.getMessage(messageKey, args,
				LocaleContextHolder.getLocale());
```

```
        }

        ...
}
```

This allows us to fetch the messages by using `LemonUtil.getMessage`, as shown below:

```
LemonUtil.getMessage("invalidSize", 6, 30)
```

The above code will return "*Size must be between 6 and 30 characters*."

mapOf

We often need to create a Java map out of some given values. For example, to pass a set of objects to the client, we might need to create a map like this:

```
{entity: "User", exception: getException()}
```

In Java, a standard way to do this would be:

```
Map<String, Object> map = new HashMap<String, Object>(2);
map.put("entity", "User");
map.put("exception", getException());
```

Too long, isn't it? Spring Lemon has a `mapOf` method, which can make it short, as below:

```
LemonUtil.mapOf("entity", "User", "exception", getException());
```

`mapOf` is coded as below:

```
@SuppressWarnings("unchecked")
public static <K,V> Map<K,V> mapOf(Object... keyValPair) {

    if(keyValPair.length % 2 != 0)
        throw new IllegalArgumentException("Keys and values must be in pairs");

    Map<K,V> map = new HashMap<K,V>();

    for(int i = 0; i < keyValPair.length; i += 2){
        map.put((K) keyValPair[i], (V) keyValPair[i+1]);
    }

    return map;
}
```

As you see, it loops through the argument array, pairs the values into map entries, and returns the map.

If you're using Guava, you can as well use its Immutable.of routines.

4 Exception handling

Throwing exceptions is a great way to break your program flow when the required conditions are not met. For example, they could be used for validating input parameters.

When an unhandled exception occurs while processing a request, your API should handle it and send a suitable error response to the client with proper HTTP code and error details. Spring MVC provides multiple ways to do it. Among those, writing a *controller advice* looks like a good way, which allows us to code the handlers globally at a single place.

What is controller advice

Controller advices are classes that can contain global exception handler methods.

> Controller advices can also contain additional things like global *ModelAttributes*.

Below is an example controller advice, which will send a *403* response to the client when an `AccessDeniedException` is thrown:

```
@RestControllerAdvice
@RequestMapping(produces = "application/json")
public class DefaultExceptionHandler {

    @ExceptionHandler(AccessDeniedException.class)
    @ResponseStatus(value = 403)
    public Map<String, Object>
       handleAuthorizationException(AccessDeniedException ex) {

         log.warn("User does not have proper rights:", ex);
         return LemonUtil.mapOf("exception", "AccessDeniedException",
                                "message", ex.getMessage());
    }
}
```

Note that the class is annotated with `@RestControllerAdvice`, and the handler method is annotated with `@ExceptionHandler(AccessDeniedException.class)` and `@ResponseStatus(value = 403)`. The return value of the handler method will become the response body, which will look as below:

```
{"exception": "AccessDeniedException", "message": "Access is Denied"}
```

Exception handling in Spring Lemon

In Spring Lemon, DefaultExceptionHandler is the controller advice class for handling exceptions. Have a look at that. Apart from `AccessDeniedException` and the generic `Exception`, it also handles a few more kinds of exceptions, which we will discuss later.

How to override exception handling

When using Spring Lemon (or your own library), if you don't find its exception handlers meeting your requirement, you can

1. Code your own controller advice class.
2. Add a `@Order(Ordered.HIGHEST_PRECEDENCE)` annotation to that class.
3. Code handler methods for those exceptions for which you want to override the response.

This StackOverflow post explains it well.

5 Application properties

Application properties, e.g. the database connection parameters, shouldn't be hardcoded in your application. They should instead be put in *properties* or *YAML* files, and injected into the application.

Spring Boot has multiple ways to have this. In this chapter, we will describe a way that we usually follow.

Segregating properties

Some of the application properties would be same across environments, while some would vary. *Spring profiles* would help segregating these.

> As you may be knowing, you can set the *spring.profiles.active* property to customize the configuration of your application. For example, you can set `spring.profiles.active=india,prod` to configure your application for using *Indian* taxation rules, in a *production* environment.

Common properties

Some of the properties, e.g. *multipart.enabled=false*, are going to be common to all environments. You can put those in an `application.properties` file, in a `config` folder inside `src/main/resources`.

> By default, Spring Boot can pick *application*.properties* files from *src/main/resources* and *src/main/resources/config*. We prefer *src/main/resources/config*, but it's just a personal choice.

Let this *application.properties* have a line like

```
# Development environment
spring.profiles.active: dev
```

This will set the default environment as `dev`. We will override it in other environments.

Environment specific properties

For each `environment` that has some environment specific properties, create an *application-`environment`.properties* in the `config` folder.

For example, to set different logging levels in each environment, you can have *application-dev.properties*, *application-test.properties* and *application-prod.properties*, as below:

application-dev.properties

```
logging.level.: INFO
```

application-test.properties

```
logging.level.: DEBUG
```

application-prod.properties

```
logging.level.: WARN
```

> You can also take a different approach: Set a default logging level in *application.properties*, and override that in these environment specific files.

Confidential properties

Some of the properties, like the database password of the production database, should not be visible to all the developers. So, don't put those in the application-prod.properties files inside your project. Instead, have those in an application-prod.properties in the directory where the JAR will be deployed, or in a *config* subdirectory therein. These external files take precedence over those in the project.

Alternatively, define environment variables (in the deployment machine) corresponding to these properties, replacing dots with underscores. For example, to set *spring.datasource.password*, you can set an environment variable `spring_datasource_password=top-secret`. Environment variables take precedence over configuration files. They are a good solution to override properties, particularly on PaaS like Pivotal Cloud Foundry, where creating external properties files would not be permitted.

Overriding the profile

You can override *spring.active.profile* using the same methods that we discussed above.

However, when writing test cases, the `@ActiveProfiles` annotation comes in handy, as below:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=RANDOM_PORT)
@ActiveProfiles("itest")
public class MyTests {
```

Injecting the properties

Now that we have discussed how to define properties, let's have a look at a good way to inject those in our code.

A simple way to inject a property in our code is to use the `@Value` annotation, like this:

```
@Value("${rememberMe.secretKey}")
private String rememberMeKey;
```

But, a cleaner way would be to define all your properties with a *prefix*, and then use a *configuration properties* class to inject those. This is helpful particularly when you are developing a library to be used by other coders.

Spring Lemon's example will make it clear.

Prefixing properties with a common word

Below is the list of properties used by Spring Lemon, which you should provide in your application. Note how each property is prefixed with `lemon`.

```
# Client web application's base URL.
# Used in the verification link mailed to the users, etc.
# Default: http://localhost:9000
lemon.application-url: http://www.example.com

# Secret string used for encrypting remember-me tokens
lemon.remember-me-key: someSecret

# Google reCAPTCHA Site and secret keys.
# Not providing these will disable captcha validation
lemon.recaptcha.sitekey: 6LdwxRcUAAAAABkhOGWQXhl9FsR27D5YUJRuGzx0
lemon.recaptcha.secretkey: 6LdwxRcUAAAAADaG0Eo1qkYCco15cnngiBoBt2IO

# Comma separated whitelisted URLs for CORS.
# Should contain the applicationURL at the minimum.
# Not providing this property would disable CORS configuration.
lemon.cors.allowed-origins: http://www.example.com,http://localhost:9000

# Methods to be allowed in CORS calls
# Default: GET,HEAD,POST,PUT,DELETE,TRACE,OPTIONS,PATCH
lemon.cors.allowed-methods: GET,POST,PUT,DELETE,OPTIONS

# Custom request headers to be allowed in CORS calls
# Default: Accept,Accept-Encoding,Accept-Language,Cache-Control,Connection,
#          Content-Length,Content-Type,Cookie,Host,Origin,Pragma,Referer,
#          User-Agent,x-requested-with,X-XSRF-TOKEN
lemon.cors.allowed-headers: x-requested-with,x-xsrf-token

# Custom response headers to be exposed to browser in CORS calls
# Default: Cache-Control,Connection,Content-Type,Date,Expires,Pragma,Server,
#          Set-Cookie,Transfer-Encoding,X-Content-Type-Options,X-XSS-Protection,
#          X-Frame-Options,X-Application-Context,X-XSRF-TOKEN
lemon.cors.exposed-headers: x-xsrf-token

# CORS maxAge
# Default: 3600L
lemon.cors.max-age: 7200L

# Properties related to the initial Admin user to be created
lemon.admin.username: admin@example.com
lemon.admin.password: admin!
```

```
# Properties to be passed to client
# should be prefixed with "lemon.shared"
lemon.shared.fooBar: 123

# Uncomment the line below
# to provide your own UserDetailsService
# lemon.enabled.user-details-service=false

# Uncomment the line below
# to disable prefixing JSON responses with ")]}',\n"
# lemon.enabled.json-prefix=false
```

Coding a configuration properties component

Prefixing the properties with `lemon` enables us to code a *configuration properties* component, like LemonProperties.java. Have a look at that and notice that

1. The class is annotated with `@ConfigurationProperties(prefix="lemon")`.
2. Default values are assigned to most properties – relieving the application developer to provide those in most cases.
3. The class and properties are commented with `/** … */` JavaDoc comment style. More about it later.
4. How comma delimited values can be injected into String arrays (e.g. see *allowedMethods*).
5. We have a property named `shared`, which is a map. That means, all the properties prefixed with *lemon.shared*, e.g. `lemon.shared.fooBar: 123` above will be put in this map. As we will see later, this will enable your applications to easily send arbitrary properties to the client.
6. The class is annotated with `@Validated`, which enables us to validate the property values by using Hibernate constraints, as below:

```
@Validated
@Configuration
@ConfigurationProperties(prefix="lemon")
public class LemonProperties {

    ...

    @Size(min=6)
    private String rememberMeKey;

    ...
}
```

Accessing the properties

Accessing the properties now becomes simple. You just have to inject the component, like this:

```
@Autowired
private LemonProperties properties;
```

How to have autocomplete

If you are using an IDE such as STS, you would experience autocomplete when editing *application.properties*. For example, see what happens when you type `spring.` there.

To have the same kind of autocomplete for your properties, you need to provide some metadata. See Spring Boot documentation for more details.

Providing such metadata manually is a tedious task. So, Spring Boot provides a `spring-boot-configuration-processor`, which generates the metadata from your *JavaDoc*, i.e. comments surrounded with `/** ... */`, as below:

```
/**
 * Google ReCaptcha Site Key
 */
private String sitekey;
```

For this, `spring-boot-configuration-processor` should be included in your build configuration file. So, if you are using maven, add it to your library's *pom.xml*:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

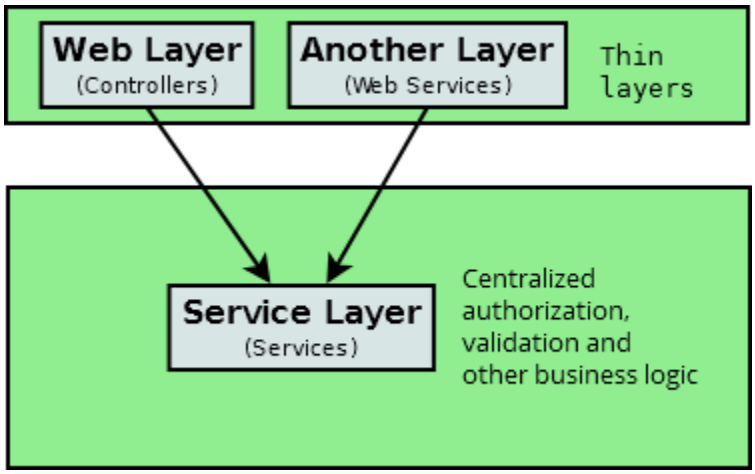
Have a look at LemonProperties to see how we have commented its properties, and the pom.xml to see how `spring-boot-configuration-processor` is included.

6 Validation

By validation, we mean validating the data received in method parameters. Spring has good support for validating method parameters not only in the controller layer, but in any layer.

Where to put the validation logic

We would prefer to put the validation logic in the service layer rather than the controller layer. This way, layers added on top will not have to bother about validation. The picture below illustrates it.



Service layer validation configurations

For doing service layer validation, Spring's MethodValidationPostProcessor could be used. It's auto-configured Spring Boot 1.5 onwards. For prior versions, you can configure it as below:

```
@Configuration
public class ValidationConfig {

    @Bean
    public MethodValidationPostProcessor methodValidationPostProcessor() {

        MethodValidationPostProcessor processor =
                    new MethodValidationPostProcessor();
        processor.setValidator(validator());
        return processor;
    }

    @Bean
    public Validator validator() {
```

```
        return new LocalValidatorFactoryBean();
    }
}
```

The Hibernate Validator is used under the hood, which is added to your project by default with *spring-boot-starter-web*.

Using validation in service classes

Once you configure a `MethodValidationPostProcessor`, you can validate your service method parameters as below:

```
@Validated
@Service
public class MyService {

      public void signup(@Valid User user) {
            ...
      }
}
```

Note the use of `@Validated` and `@Valid`, because of which, when the `signup` method will be called, constraint annotations in the `User` class will be checked, and the `user` object will be validated accordingly. In case of any errors, a ConstraintViolationException will be thrown. We need to catch that exception and return some error response to the client accordingly.

How to catch constraint violation exceptions

The best way to catch constraint violation exceptions would be to code a handler method in our controller advice, like this:

```
@RequestMapping(produces = "application/json")
@ExceptionHandler(ConstraintViolationException.class)
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public @ResponseBody Map<String, Object>
handleConstraintViolationException(ConstraintViolationException ex) {
      // return the errors to the client in the response body
}
```

Whenever a ConstraintViolationException will be thrown, we will receive the exception in the `ex` parameter of the above method. We will then need to extract the error data, and return it. Spring will convert the return data to JSON and put that in the response. The response code will be 400, because of the annotation `@ResponseStatus(value = HttpStatus.BAD_REQUEST)` above.

How to return the errors

How to get the validation errors

ConstraintViolationException's *getConstraintViolations()* method returns the errors as a set of ConstraintViolations. Our job then would be to transform `ex.getConstraintViolations()` into a simple structure and return that, which will be converted to JSON and put in the response.

Structure to send those to the client

A simple structure for returning the validation errors to the client could look like this:

```
[
    {
      field: "user.email",
      code: "com.naturalprogrammer.spring.invalid.email"
      message: "Not a well formed email address"
    },
    {
      field: "user.email",
      code: "com.naturalprogrammer.spring.blank.email"
      message: "Email needed"
    },
    {
      field: "user.name",
      code: "blank.name"
      message: "Name required"
    },
    {
      field: "user", // OR null OR "something.unknown"
      code: "some.code"
      message: "Some form level error"
    },
    {
      field: "user.address.lane",
      code: "com.naturalprogrammer.spring.long.lane"
      message: "Lane must be less than 50 characters"
    }
]
```

This is an array of error messages. Each error message has

1. A `field` containing the *path* of the field. Nested fields can have multiple dots in them, like `"user.address.lane"` in the fifth case above. In case the error is not related to a specific field, but is a form-level global error, as in the fourth case above, `field` can be just the object name, `null` , or something unknown.
2. A `code` containing the error code.
3. A `message` containing the internationalized error message.

The error messages in the above array can be structured as a simple class like this:

```
public class FieldError {

      private String field;
      private String code;
      private String message;

    ... getters and other utility methods ...
}
```

Converting ConstraintViolations to FieldErrors

A `ConstraintViolation` has the following attributes:

```
{
   propertyPath: "signup.user.email",
   messageTemplate: "error-message-code",
   message: "Not a valid email"
   ...
}
```

The `propertyPath` above is of the format *`methodName.objectName.fields`*. We won't need the *`methodName`* prefix when sending it to the client. To strip it out, the code snippet below can be used:

```
String field = StringUtils.substringAfter(
      constraintViolation.getPropertyPath().toString(), ".");
```

So, in summary, the method below would convert a `ConstraintViolation` to a `FieldError`:

```
private static FieldError of(ConstraintViolation<?> constraintViolation) {

      String field = StringUtils.substringAfter(
                        constraintViolation.getPropertyPath().toString(), ".");

      return new FieldError(field,
                  constraintViolation.getMessageTemplate(),
                  constraintViolation.getMessage());
}
```

A set of `ConstraintViolation`s can then be converted to a list of `FieldError`s using *Java 8* and the above `of` method, this way:

```
public static List<FieldError> getErrors(
             Set<ConstraintViolation<?>> constraintViolations) {

      return constraintViolations.stream()
                  .map(FieldError::of).collect(Collectors.toList());
}
```

Our exception handler can use this method to return the errors. To see the exact details, look at the FieldError and the DefaultExceptionHandler classes in Spring Lemon.

How to use method validation

Method validation can be used in multiple ways. See the three examples below:

```
@Validated
@Service
public class MyService {

	public void firstMethod(@Valid User user) {
		...
	}

	public void secondMethod(@Valid @Email @NotBlank String email) {
		...
	}

	@Validated(SignUpValidation.class)
	public void thirdMethod(@Valid User user) {
		...
	}

}
```

In the first method, we are validating a `User` object. In the second method, we are validating a primitive string. In the third method, we are providing a constraint group by using the `@Validated(SignUpValidation.class)` annotation on the method. That is telling Spring to validate only those fields that have the `SignUpValidation.class` in their groups, like this:

```
public class User {

    @Size(min=1, max=30, groups = {SignUpValidation.class, UpdateValidation.class})
    private String name;

}
```

The `SignUpValidation.class` is nothing but just a marker interface. As you shall see in later chapters, such constraint grouping is going to be quite useful.

Custom validation

Hibernate validator has a nice set of built-in constraints. But, you are also going to need validations beyond that. For example, when signing up, you will need to check the uniqueness of the email id.

We recommend a couple of ways to do such custom validations, which are discussed next.

Creating custom constraints

We can easily create custom constraint annotations and write validation logic for those. For example, let's see how to code a `@UniqueEmail` constraint, which we are going to use for signing up, in our `AbstractUser` class, like this:

```
@UniqueEmail
protected String email;
```

Creating the annotation

Custom annotations can be coded like this:

```
@NotBlank(message = "{com.naturalprogrammer.spring.blank.email}")
@Size(min=4, max=250)
@Email(message = "{com.naturalprogrammer.spring.invalid.email}")
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,ElementType.FIELD,ElementType.CONSTRUCTOR,
         ElementType.PARAMETER,ElementType.ANNOTATION_TYPE})
@Constraint(validatedBy=UniqueEmailValidator.class)
public @interface UniqueEmail {

    String message() default "{com.naturalprogrammer.spring.duplicate.email}";
    Class[] groups() default {};
    Class[] payload() default {};
}
```

As you see, it's annotated with a few other constraints, namely `@NotBlank`, `@Size` and `@Email`. In addition, it's also annotated with `@Constraint(validatedBy=UniqueEmailValidator.class)`.

That means, fields annotated with this `@UniqueEmail` annotation will be validated for *@NotBlank*, *@Size* and *@Email*. In addition, the code inside `UniqueEmailValidator` will also be used, which we discuss next.

Coding the validator

Validator classes, such as the `UniqueEmailValidator` referred above, look like this:

```
@Component
public class UniqueEmailValidator
implements ConstraintValidator<UniqueEmail, String> {

	@Override
	public void initialize(UniqueEmail constraintAnnotation) {
		// can access the attribute values of the annotation
		// and store as instance variables
	}

	@Override
	public boolean isValid(String email, ConstraintValidatorContext context) {
		// validation logic goes here
```

```
	}
}
```

To know further about creating custom constraints, read this.

Coming to Spring Lemon, its UniqueEmailValidator looks like this:

```
@Component
public class UniqueEmailValidator
implements ConstraintValidator<UniqueEmail, String> {

	@Autowired
	private AbstractUserRepository<?,?> userRepository;

	@Override
	public void initialize(UniqueEmail constraintAnnotation) {
	}

	@Override
	public boolean isValid(String email, ConstraintValidatorContext context) {

		return !userRepository.findByEmail(email).isPresent();
	}
}
```

Combining existing constraints

If you want just to combine a few existing constraints, and don't need to code any new validation logic, leave the `validatedBy` blank. Here is the @Password example from Spring Lemon:

```
@NotBlank(message="{com.naturalprogrammer.spring.blank.password}")
@Size(min=6, max=30,
	message="{com.naturalprogrammer.spring.invalid.passwordSize}")
@Target({ METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
@Documented
@Constraint(validatedBy = { })
public @interface Password {

	String message() default "{com...invalid.passwordSize}";
	Class<?>[] groups() default { };
	Class<? extends Payload>[] payload() default { };
}
```

Class level custom constraints

Both the `@UniquePassword` and `@Password` constraints, which we discussed above, can validate single fields. But, sometimes your validations may need to access more than one field. For example, when a user wishes to change his password, you can show him a form having two fields - *password* and *retype password*. On submit, you would check that both the values are same.

Such validations need us to code *class level constraints*. Let's see how to code a *retype password* validator.

Creating the annotation

Creating the annotation is no different:

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,
              ElementType.METHOD,
              ElementType.FIELD,
              ElementType.CONSTRUCTOR,
              ElementType.PARAMETER,
              ElementType.ANNOTATION_TYPE})
@Constraint(validatedBy=RetypePasswordValidator.class)
public @interface RetypePassword {

    String message() default "{com.naturalprogrammer.spring.different.passwords}";

    Class[] groups() default {};

    Class[] payload() default {};
}
```

Coding the validator

Coding the validator would be a bit different:

```
@Component
public class RetypePasswordValidator
implements ConstraintValidator<RetypePassword, RetypePasswordForm> {

      @Override
      public boolean isValid(RetypePasswordForm retypePasswordForm,
            ConstraintValidatorContext context) {

            if (!Objects.equals(retypePasswordForm.getPassword(),
                                    retypePasswordForm.getRetypePassword())) {

                  context.disableDefaultConstraintViolation();
                  context.buildConstraintViolationWithTemplate(
                        "{com.naturalprogrammer.spring.different.passwords}")
                  .addPropertyNode("retypePassword").addConstraintViolation();

                  return false;
            }
            return true;
      }

      @Override
      public void initialize(RetypePassword constraintAnnotation) {
      }
```

```
}
```

Note that

1. In the declaration, we are using `RetypePasswordForm`, which is a class rather than a simple field. It is the *type* of the data that would be validated in the `isValid` method. A good practice would be to have this as an interface, and let all your forms needing this validation implement this interface.
2. When the validation will fail, by default an object level error will be thrown. The code highlighted above is needed to make it specific to a field instead.

Look at RetypePassword.java, RetypePasswordForm.java and RetypePasswordValidator.java of Spring Lemon for complete source code.

Captcha validation

We should avoid a hacker writing a script and doing automated entries, in forms like signup. The new reCAPTCHA by Google is a great solution to this.

How to use reCAPTCHA

Google documentation has details about how to use reCAPTCHA. Specifically,

1. Get a pair of *site key* and *secret key*, by registering your sites with Google. Note that
    a. Multiple sites can use the same pair of keys – you just need to let Google know the sites, either at the time of registering or later.
    b. No need to add *localhost* – it's added by default.
2. Integrate it on your form at the client-side. Although google has generic instructions about it, for AngularJS, angular-recaptcha works great.
3. At the client side, when a user solves the captcha, a *response token* is generated. On submit, send the *token* to the server side along with the business data.
4. At the server side, POST the received token along with the *secret key* to *https://www.google.com/recaptcha/api/siteverify*. The response, as detailed here, will tell whether the validation succeeded or failed. It would be good to create a custom constraint for this.

Creating a @Captcha constraint

Creating the `@Captcha` annotation would be simple

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD,
                ElementType.FIELD,
                ElementType.CONSTRUCTOR,
                ElementType.PARAMETER,
                ElementType.ANNOTATION_TYPE})
@Constraint(validatedBy=CaptchaValidator.class)
public @interface Captcha {
```

```
    String message() default "{com.naturalprogrammer.spring.wrong.captcha}";
    Class[] groups() default {};
    Class[] payload() default {};
}
```

Doing the validation

Let's now see how the validator class can be written.

As you know, its `isValid` method would do the validation. The steps to follow could be:

1. Have the *site key* and *secret key* as application properties. They can be provided like this:

```
lemon.recaptcha.sitekey: 6LdwxRcUAAAAABkhOGWQXhl9FsR27D5YUJRuGzx0
lemon.recaptcha.secretkey: 6LdwxRcUAAAAADaG0Eo1qkYCco15cnngiBoBt2IO
```

2. *Check whether captcha is disabled*. Sometimes, e.g. when doing automated integration testing or quick prototyping, one may like to disable captcha validation. So, let's say that, if there is no *sitekey* property, we just pass the validation:

```
@Autowired
private LemonProperties properties;

@Override
public boolean isValid(String captchaResponse,
      ConstraintValidatorContext context) {

      if (properties.getRecaptcha().getSitekey() == null)
          return true;
      ...
```

3. *Ensure that captchaResponse is not blank*. Apache StringUtils can be used for it:

```
      if (StringUtils.isBlank(captchaResponse))
           return false;
```

4. *Create a* MultiValueMap of the *captchaResponse* and the *secretKey*, for posting to *https://www.google.com/recaptcha/api/siteverify*:

```
MultiValueMap<String, String> formData =
      new LinkedMultiValueMap<String, String>(2);
formData.add("response", captchaResponse);
formData.add("secret", properties.getRecaptcha().getSecretkey());
```

We need a *MultiValueMap* rather than a normal *HashMap* so that Spring MVC can convert it to HTML form data using a FormHttpMessageConverter, which comes pre-configured with Spring MVC.

5. *POST https://www.google.com/recaptcha/api/siteverify*: We can use a RestTemplate for doing the POST:

```
ResponseData responseData = restTemplate.postForObject(
   "https://www.google.com/recaptcha/api/siteverify",
   formData, ResponseData.class);
```

Note that we are receiving the response in a `ResponseData` object, which would look like this:

```
private static class ResponseData {

      private boolean success;

      @JsonProperty("error-codes")
      private Collection<String> errorCodes;

      // getters and setters
}
```

The *RestTemplate* object that we have used can be created in any manner. In Spring Lemon, we have it as a bean in the LemonConfig class:

```
@Bean
@ConditionalOnMissingBean(RestTemplate.class)
public RestTemplate restTemplate() {

      return new RestTemplate();
}
```

This configures a *RestTemplate* if one is not already configured in your application.

6. *Finally, check whether google returned a success*:

```
if (responseData.success)
    return true;

return false;
```

Look at Spring Lemon's CaptchaValidator.java for the complete source code.

Validating inside service methods

So far we have been discussing about annotation-based validation. Another way of validation could just be checking the data straight inside your service methods and throwing exceptions if invalid data is found. A ControllerAdvice method could be used for processing the exceptions.

Next chapter will cover this in details.

7 Multi-error exceptions

As you know, we can throw exceptions from our code, and write controller advices to process those. This mechanism could be used for validating data. But, instead of eagerly throwing an exception when the first validation fails, sometimes you we may like to check for multiple validations, and then throw a cumulative exception. For example, when validating input parameters, you might like to accumulate all the errors first, and then throw the exception.

In this chapter, we are going to layout a decent pattern for throwing such multi-error exceptions.

Coding a custom exception class

Let's first code a custom exception class, named `MultiErrorException`, for holding the error messages. Our existing FieldError class can be used for holding an error message. So, `MultiErrorException` will have a list of `FieldError`s:

```
public class MultiErrorException extends RuntimeException {

	private List<FieldError> errors = new ArrayList<FieldError>(10);

	public List<FieldError> getErrors() {
		return errors;
	}

	...
}
```

How to use the class

You can use `MultiErrorException` in the service methods by creating an instance of it, go on populating its *errors* list if there are errors, and then throw it. For example, your code could look like this:

```
MultiErrorException exception = new MultiErrorException();
exception.check("email", user.getEmail() != null, "emailNull")
         .check("name", user.getPassword() != null, "passwordNull")
         .go();
```

The above code uses a `check` method, which we are yet to discuss, which takes three parameters. The first one is the name of the field, the second a *condition*, and the third a *messageKey*. If the condition will fail, a new *FieldError* will be added to the *errors* list. The error message will be extracted from your *messages.properties* file by using the given *messageKey*.

The `go` method will throw the exception, only if there are errors present in the list.

Coding *check* and *go*

The code for the above `check` and `go` could look like this:

```
public MultiErrorException check(String fieldName, boolean valid,
            String messageKey, Object... args) {

      if (!valid)
            errors.add(new FieldError(fieldName,
                  LemonUtil.getMessage(messageKey, args)));

      return this;
}

public void go() {
      if (errors.size() > 0)
            throw this;
}
```

LemonUtil.getMessage extracts the message from the *messages.properties* file, as we have seen earlier.

If you are dealing with an error not pertaining to a specific field, you can pass `null` as the field. To make it handy, how about we create another check method, like this:

```
public MultiErrorException check(boolean valid,
            String messageKey, Object... args) {

      return check(null, valid, messageKey, args);
}
```

Overriding *getMessage*

As you know, in Java, `Throwable` is the base class for all the exceptions. It has a `getMessage` method for returning a short error message. We should override that. The first error message from our list could just be returned in the overridden method:

```
@Override
public String getMessage() {

      if (errors.size() == 0)
            return null;

      return errors.get(0).getMessage();
}
```

A couple of utility methods

We would need to create a new `MultiErrorException` everytime we are going to do the checks, as below:

```
MultiErrorException exception = new MultiErrorException();
exception.check(...)
         .check(...)
         .go();
```

Why not put the first couple of lines in an utility method, like this:

```
public static MultiErrorException check(
            String fieldName, boolean valid, String messageKey, Object... args) {
      return new MultiErrorException().check(fieldName, valid, messageKey, args);
}
```

For the cases where the field would be null, we can have another handy utility method:

```
public static MultiErrorException check(
            boolean valid, String messageKey, Object... args) {
      return check(null, valid, messageKey, args);
}
```

We have these in the LemonUtil class in Spring Lemon. Using these, your code now would look cleaner:

```
LemonUtil.check(..).check(..)...go();
```

Using MultiErrorException along with Java 8

Sometimes, particularly when using Java 8, we would pass an instance of the exception instead of explicitly throwing it. Consider this example:

```
U user = userRepository.findByEmail(email)
      .orElseThrow(() -> {
            MultiErrorException exception = new MultiErrorException();
            exception.errors.add(new FieldError(fieldName, messageKey,
                        LemonUtil.getMessage(messageKey)));
            return exception; // don't throw!
      });
```

The above `findByEmail` method returns an Optional, which needs an exception returned from the *orElseThrow* block. So, we have created a *MultiErrorException* and returned it. Instead of writing so much code each time, it would be best to have a static method in the *MultiErrorException* class for this:

```
public static Supplier<MultiErrorException> supplier(String fieldName,
```

```
            String messageKey, Object... args) {

      MultiErrorException exception = new MultiErrorException();
      exception.errors.add(new FieldError(fieldName, messageKey,
                  LemonUtil.getMessage(messageKey, args)));

      return () -> exception;
}
```

The `orElseThrow` block would now be simpler, looking like this:

```
U user = userRepository.findByEmail(email)
            .orElseThrow(MultiErrorException.supplier("email",
                  "com.naturalprogrammer.spring.userNotFound"));
```

Finally, we can code one another static `of` method, which would be handy when we don't need to supply a field:

```
public static Supplier<MultiErrorException> supplier(String messageKey,
      Object... args) {

      return MultiErrorException.supplier(null, messageKey, args);
}
```

So, this completes our `MultiErrorException` class. You can view its complete source code in Spring Lemon here.

Handling the exception

You already know that handling these exceptions is going to be easy. We just need a method in our controller advice class, as below:

```
@ExceptionHandler(MultiErrorException.class)
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public Map<String, Object>
            handleMultiErrorException(MultiErrorException ex) {

      List<FieldError> errors = ex.getErrors();

      return LemonUtil.mapOf("exception", "MultiErrorException",
                  "message", ex.getMessage(), "errors", errors);
}
```

LemonUtil.mapOf above, as discussed earlier, is a utility method for building a map from the given arguments.

So, the client is going to receive a response looking like this:

```
{
  exception: "MultiErrorException",
```

```
  message: "Name required "
  errors: [
    {
      field: "user.name",
      code: "blank.name"
      message: "Name required"
    },
    {
      field: "user", // OR null OR "something.unknown"
      code: "some.code"
      message: "Some form level error"
    },
    {
      field: "user.address.lane",
      code: "com.naturalprogrammer.spring.long.lane"
      message: "Lane must be less than 50 characters"
    }
  ]
}
```

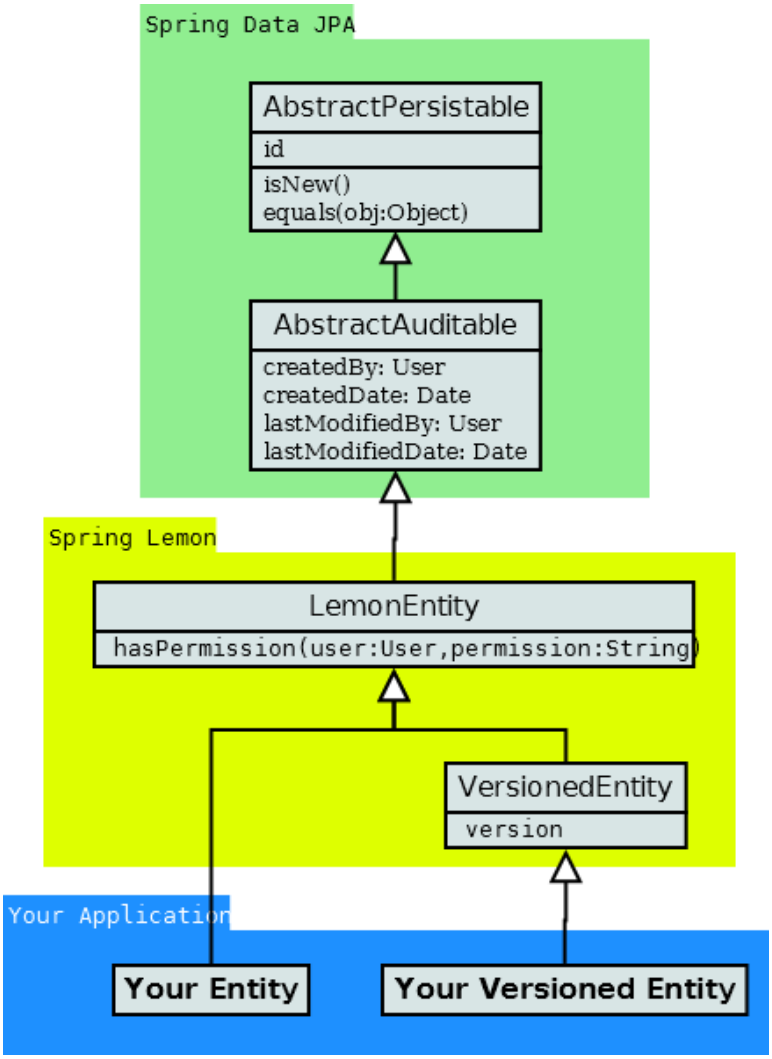
That's all! Have fun in your service methods by using clean validation constructs like this:

```
LemonUtil.check(..).check(..)...go();
```

It's used at many places in Spring Lemon's service methods, which we will discuss later.

8 JPA Domain Model

Although JPA entity classes can be created just by annotating POJO classes with `@Entity`, it helps to have some base classes for holding the common elements. In this chapter, we are going to look at how we have designed these base classes in Spring Lemon. The diagram below illustrates it.



Let's look at these classes more closely.

AbstractPersistable

AbstractPersistable, provided by Spring Data JPA, is the top-most in the hierarchy. It contains the primary key and a few useful methods. Its `equals` method is very handy for comparing entities.

AbstractAuditable

AbstractAuditable, again provided by Spring Data JPA, adds columns for containing audit data, e.g. *who created* and *who updated* the entity. Spring Data JPA populates those automatically, provided we have the following:

1. An `orm.xml` in the *META-INF* folder, just like we have in Spring Lemon at src/main/resources/META-INF/orm.xml.
2. A component implementing the AuditorAware interface. It is used by Spring Data JPA to fetch the currently logged in user.

Here is how `AuditorAware` is implemented in Spring Lemon:

```
@Component
public class AuditorAwareImpl
      <U extends AbstractUser<U,ID>,
       ID extends Serializable>
implements AuditorAware<U> {

      @Override
      public U getCurrentAuditor() {
            return LemonUtil.getUser();
      }
}
```

The Java generics in the class declaration was needed to have an extensible user entity. Do not focus on it now; the next chapter will cover it in details.

The overridden `getCurrentAuditor()` returns the currently logged in user. It uses `LemonUtil.getUser()`, which we will cover in the security chapter.

Here is the complete source code, for your reference.

Spring Data JPA reference material has covered JPA auditing in more details.

LemonEntity

LemonEntity extends *AbstractAuditable* and adds a `hasPermission` method. `hasPermission` is used for checking whether a *user* has a certain *permission* on the entity object. It looks like this:

```
public boolean hasPermission(U user, String permission) {
      return false;
}
```

Override this method in your entity classes to specify if the given `user` has the given `permission` on the entity. For example, if you have an Employee entity which can be viewed by anybody logged in, but edited only by HR manager, you could override `hasPermission` like this:

```
@Override
public boolean hasPermission(User user, String permission) {

	if ("view".equals(permission))			// can be viewed by
		return user != null;			// anybody logged in

	if ("edit".equals(permission))			// can be edited by
		return user != null			// a logged in user
		    && user.hasRole("HR_MANAGER");	// who is an HR manager

	return false;
}
```

We will see live examples of this in later chapters.

The entities that do not need versioning and optimistic locking can extend this class.

VersionedEntity

VersionedEntity extends `LemonEntity` and adds optimistic locking capabilities. It has a version column:

```
@Version
private long version;
```

The `@Version` annotation tells JPA to increment this column every time when the entity is updated in the database. Before an update, the JPA provider checks this column in the database to find out whether it is already incremented, i.e. someone else has already updated the entity concurrently. An exception is thrown if this column is already incremented, thus preventing concurrent modifications.

The `version` column is for JPA's internal use, and you aren't allowed to alter it. But, you can use it as below:

1. When a user visits a page to update an entity, he is first shown a form filled in with the original data. When sending that original data from the server, you should send its *version* along with it.
2. The user then alters the data and presses *submit*. That time, you should attach the *version* along with the modified data.
3. At the server side, you should then retrieve a fresh copy of the entity from the database and check whether its version has increased meanwhile. If the version has increased, someone else must have updated it meanwhile. You should then throw an exception. If the version has not increased, no one else has touched it, and so go ahead with the

update. (If you are wondering why would JPA not check it automatically, read this stackoverflow post)

4. The exception thrown above should be propagated to the client. The user might then like to redo the update, after fetching the new version of the entity.

In Spring Lemon, we have a VersionException to be thrown in such cases. It looks like this:

```
public class VersionException extends RuntimeException {

	public VersionException(String entityName) {
		super(LemonUtil.getMessage(
			"com.naturalprogrammer.spring.versionException", entityName));
	}
}
```

It would require you to define `com.naturalprogrammer.spring.versionException` in your *messages.properties*, say like this:

```
com.naturalprogrammer.spring.versionException: Could not save. Looks like the {0} is already modified by somebody else. Please refresh and try again.
```

How to throw the exception

As discussed above, in your code, you should check if the versions match, and if not, throw a `VersionException`, like this:

```
if (freshEntityFromDb.getVersion() != modifiedEntity.getVersion())
	throw new VersionException(freshEntityFromDb.getClass().getSimpleName());
```

In Spring Lemon, we have a handy utility method in LemonUtil which you can use instead:

```
public static <U extends AbstractUser<U,ID>, ID extends Serializable>
void validateVersion(VersionedEntity<U,ID> original, VersionedEntity<U,ID> updated) {

	if (original.getVersion() != updated.getVersion())
		throw new VersionException(original.getClass().getSimpleName());
}
```

Do not bother about the generics at the moment; it will be elaborated in the next chapter. So, using the above method, checking versions becomes simple:

```
LemonUtil.validateVersion(freshEntityFromDb, modifiedEntity);
```

How to pass the exception to the client

We can use a controller advice method to catch the exception and pass it to the client:

```
@RestControllerAdvice
@RequestMapping(produces = "application/json")
```

```
public class DefaultExceptionHandler {

    @ExceptionHandler({VersionException.class})
    @ResponseStatus(value = HttpStatus.CONFLICT)
    public Map<String, Object> handleVersionException(VersionException ex) {

       return LemonUtil.mapOf("exception", "VersionException",
                              "message", ex.getMessage());
    }
}
```

This will send a *409 Conflict* response to the client, with the returned data as JSON.

In Spring Lemon, we have this code in our DefaultExceptionHandler class.

Your Entities

Your entities can extend from either `LemonEntity` or `VersionedEntity`. For example, if you want to create an `Employee` entity, it could look like this:

```
public class Employee extends VersionedEntity<User, Long> {
       ...
}
```

Notice the use of Java generics above. We needed to pass `User` and `Long`, indicating that our primary key is `Long`, and user type is `User`. The `User` type is used in `AbstractAuditable` for holding audit data, if you recall.

If you want to avoid the "long looking" generics in the class declaration of your entities, just create your own intermediate classes, like this:

```
public abstract class MyVersionedEntity extends VersionedEntity<User, Long> {

}
```

Should I use optimistic locking?

We see no reason you shouldn't. We recommend that all your entities should extend from the `VersionedEntity` instead of just the `LemonEntity`.

9 The User Entity

The User entities of all your applications will have many properties and methods in common. Those could be put in a base class. In Spring Lemon, we have an `AbstractUser` class holding such common properties and methods. In this chapter, let's cover *AbstractUser* in details.

AbstractUser

The declaration of `AbstractUser` looks like this:

```
@MappedSuperclass
public abstract class AbstractUser
<U extends AbstractUser<U,ID>, ID extends Serializable>
      extends VersionedEntity<U, ID> {
    ...
```

The generics used above looks a bit complex, but is not as untidy as it looks. The first parameter, `U`, represents the concrete user class, which would be defined in your application. Recall that it is needed in `AbstractAuditable` for storing the user who created and the user who updated the entity. As you see above, it is written as `U extends AbstractUser...`, which means that your User class is going to inherit the `AbstractUser` class.

The second parameter, `ID`, is the type of the primary key.

Your concrete user class

The declaration of the concrete user class in your application can look like:

```
@Entity
@Table(name="usr")
public class User extends AbstractUser<User, Long> {
    ...
```

The generic parameters of `AbstractUser` above are

1. `User` which is the concrete type of your user, and
2. `Long` which is the type of the primary key.

Columns in AbstractUser

So, what columns should go in `AbstractUser`? Those that would be common to all your applications. In Spring Lemon, we have the following:

Email

A mandatory email id of the user, declared like this:

```
@JsonView(SignupInput.class)
@UniqueEmail(groups = {SignUpValidation.class})
@Column(nullable = false, length = EMAIL_MAX)
protected String email;
```

> Spring Lemon uses this as the *login id*. But, if you want, you can have a separate *username* column in your concrete user class, and use that as the login id instead.

The `@JsonView` annotation above is a part of an important measure to prevent malicious users from injecting extra fields into the User object. See the explanation in the signup handler for more details.

Note that we are using the `UniqueEmail` constraint that we have discussed earlier. We have provided it a `SignUpValidation.class` group, which will let the validation be done only when the service method would be annotated with `@Validated(SignUpValidation.class)`, like this:

```
@Validated(SignUpValidation.class)
public void signup(@Valid U user) {
```

`SignUpValidation.class` is nothing but a marker interface:

```
public interface SignUpValidation {}
```

Password

A mandatory password, which the user will use while logging in. It is declared like this:

```
@JsonView(SignupInput.class)
@Password(groups = {SignUpValidation.class, ChangeEmailValidation.class})
@Column(nullable = false) // no length because it will be encrypted
protected String password;
```

We haven't provided a *length* in the `@Column` annotation, because we should store passwords encrypted, and after encrypting the length would vary.

Roles

A set of Strings, e.g. {"UNVERIFIED", "ADMIN"}, declared like this:

```
@ElementCollection(fetch = FetchType.EAGER)
private Set<String> roles = new HashSet<String>();
```

`roles` would determine the authority of the user, i.e. which parts of the application the user will have access to.

In Spring Lemon, we thought to provide built-in support for the following roles:

UNVERIFIED

When a user's mail is not verified yet, the user would have a UNVERIFIED role.

BLOCKED

If a user misuses the application, he would be BLOCKED.

ADMIN

An ADMIN user that is not UNVERIFIED or BLOCKED would be able to do many admin operations, such as BLOCKing other users.

In Spring Lemon, the Role interface defines these as constants:

```
public static interface Role {

	static final String UNVERIFIED = "UNVERIFIED";
	static final String BLOCKED = "BLOCKED";
	static final String ADMIN = "ADMIN";
}
```

Verification and forgot password codes

These fields would help in the *verification* and *forgot password* process, as we will see later.

```
@Column(length = UUID_LENGTH)
protected String verificationCode;

@Column(length = UUID_LENGTH)
protected String forgotPasswordCode;
```

New email

If a user would like to change his email, before doing the change, we woud first verify his new email id by sending a mail. Till the verification is done, his new email will be stored in a `newEmail` field, looking like this:

```
@UniqueEmail(groups = {ChangeEmailValidation.class})
@Column(length = EMAIL_MAX)
protected String newEmail;
```

Change email code

`changeEmailCode` is used for storing a secret code that's mailed to the user in the verification mail. It's declared like this:

```
@Column(length = UUID_LENGTH, unique=true)
```

```
protected String changeEmailCode;
```

We will discuss about change email functionality in details later.

Transient properties

Apart from the above columns, AbstractUser also contains many useful transient properties, which would not be stored in the database. These are discussed next.

Captcha response

Used for holding the captcha response received from the front-end while signing up. It is declared like this:

```
@Transient
@JsonView(SignupInput.class)
@Captcha(groups = {SignUpValidation.class})
private String captchaResponse;
```

Computed properties

There are many handy computed properties in AbstractUser:

```
@Transient
protected boolean unverified = false;

@Transient
protected boolean blocked = false;

@Transient
protected boolean admin = false;

@Transient
protected boolean goodUser = false;

@Transient
protected boolean goodAdmin = false;

@Transient
protected boolean editable = false;

@Transient
protected boolean rolesEditable = false;

@Transient
protected Collection<GrantedAuthority> authorities;
```

`unverified`, `blocked` and `admin` correspond to roles of the user in the roles set. A `goodUser` is one who is not *unverified* or *blocked*. A `goodAdmin` is one who is an *admin* and a *goodUser* as well.

`editable` represents whether the currently logged-in user can edit this entity. In Spring Lemon, we follow the business rule: *a user can be edited only if he himself or a good admin has logged in*.

`rolesEditeble` represents whether the logged-in user can alter the roles of this entity. In Spring Lemon, we follow the business rule: *Only a good admin can alter the roles of users. He can't alter his own roles, though.*

`authorities` holds the Spring Security authorities of a user – we will discuss about it in the Security chapter.

These are populated using a helper method, discussed next.

Helper methods

We have many helper methods in AbstractUser, as given below.

hasRole

Checks if a user has a certain role:

```
public final boolean hasRole(String role) {
	return roles.contains(role);
}
```

decorate

Sets the transient properties of the user. Needs the currently logged in user as a parameter. Looks like this:

```
public U decorate(U currentUser) {

	unverified = hasRole(Role.UNVERIFIED);
	blocked = hasRole(Role.BLOCKED);
	admin = hasRole(Role.ADMIN);
	goodUser = !(unverified || blocked);
	goodAdmin = goodUser && admin;

	editable = false;
	rolesEditable = false;

	if (currentUser != null) {
		editable = currentUser.isGoodAdmin() || equals(currentUser);
		rolesEditable = currentUser.isGoodAdmin() && !equals(currentUser);
	}

	computeAuthorities();

	return (U) this;
}
```

Note how the flags are set:

1. *goodUser* is one who is neither unverified nor blocked.
2. A goodUser and admin is called a *goodAdmin*.
3. A user is *editable* (by the currentUser) if the currentUser is either a goodAdmin or editing himself.
4. Roles are editable only by goodAdmins. That too, one can't edit one's own roles.

Also note that `computeAuthorities()` is called to set the authorities. Wait until the Security chapter to know more about it.

There is another decorate method, which calls the above one, passing it the currently logged in user. It looks like this:

```
public U decorate() {
	return decorate(LemonUtil.getUser());
}
```

LemonUtil.getUser() above returns the currently logged in user, as we shall see in the Security chapter.

hideConfidentialFields

Before sending the user data to the client, we should hide the confidential fields. This method does that:

```
public void hideConfidentialFields() {

	setCreatedDate(null);
	setLastModifiedDate(null);
	password = null;
	verificationCode = null;
	forgotPasswordCode = null;

	if (!editable)
		email = null;
}
```

hasPermission

Overrides hasPermission of LemonEntity, for checking if the currently logged-in user has permission to access this user. In the Security chapter, we will discuss how this method would be used.

```
@Override
public boolean hasPermission(U currentUser, String permission) {

	decorate(currentUser);

	if (permission.equals("edit"))
```

```
            return editable;

        return false;
}
```

setIdForClient

Your client application would be needing the current-user data often. But, it won't need all the fields. Instead, only a few, like *id*, *name* and *roles* will be needed. So, instead of passing the entire current-user object, we can create a new object, copy only the needed fields, and pass that.

But, we won't be able to set the id of the new object like this:

```
newUser.setId(currentUser.getId()); // won't work
```

The above will give error because `setId` is not a public method. Hence, `setIdForClient` can be used to set the id, which looks like this:

```
public void setIdForClient(ID id) {
        setId(id);
}
```

Give a look at AbstractUser.java - that will give you more insight. Apart from the ones discussed above, you will also find some more attributes and methods there, which will be covered later in the security chapter.

AbstractUserRepository

We would need a Spring Data repository interface for accessing the user records, as you would be knowing. Spring Lemon has a base class for it, like this:

```
@NoRepositoryBean
public interface AbstractUserRepository
        <U extends AbstractUser<U,ID>, ID extends Serializable>
extends JpaRepository<U, ID> {

        Optional<U> findByEmail(String email);
        Optional<U> findByForgotPasswordCode(String forgotPasswordCode);
}
```

Note that we have annotated it with `@NoRepositoryBean`. By this, we are telling Spring Data not to provide us an implementation for this interface. Instead, in your application, you are supposed to extend it like this:

```
public interface UserRepository extends AbstractUserRepository<User, Long> {
    // any more methods you need
}
```

Use of Java 8 Optional

Note how `findByEmail` and `findByForgotPasswordCode` return Java 8 *Optional*. It enables us to write code like this:

```
U user = userRepository.findByEmail(email)
                       .orElseThrow(MultiErrorException.supplier(
                          "com.naturalprogrammer.spring.userNotFound"));
```

The above code returns a user with the given email, if found. Otherwise, an exception is thrown.

10 Security

Security, as you know, is broadly divided into two areas: *Authentication* and *Authorization*. In layman terms, *Authentication* means logging the users in, whereas *Authorization* means restricting access based on the rights of the users.

In this chapter, we will discuss how these could be configured for a REST API by using Spring Security, and how exactly Spring Lemon does it.

Authentication

There are numerous ways to have authentication in applications. Among those, the *form/cookie based authentication* is the most widely used one. In simple terms, it asks for a username and password, and then authenticates the user for a session (or for a longer period on the same computer if "Remember Me" is checked). A cookie, typically named *JSESSIONID*, is used for identifying the session.

Form based authentication depends on having a session state at the server side. But, by definition, pure RESTful APIs should be stateless. So, you might conclude that form based authentication isn't a good choice for securing REST APIs. In fact, to become stateless, some people seem to avoid form based authentication and go for OAuth2, JWT or custom token authentication.

In contrast, in Spring Lemon we thought to support form based authentication first, and the current version of Spring Lemon has just that. We have plans to support more kinds in later versions, though.

"Why?" you may ask, "Why not choose a stateless approach right from the beginning?"

Well, in practice, having a secure stateless API seems complex. Without repeating what has been already said about it, let us just point to a couple of authentic sources:

1. It very definitely is a Good Thing to use the session for authentication and CSRF protection
2. The State of Securing RESTful APIs with Spring

So, being stateful won't be bad, we thought. Form based authentication then seemed like a good first choice, instead of going for a more complex solution. It's also the most widely supported solution among browser based web frameworks.

> If you intend to call your APIs not only from browsers, but also from other applications, remember to the send JSESSIONID etc. along with your calls. It is illustrated in the Testing chapter, where we will see how to use *RestAssured* to call our API.

Enough theory. Let's now see how to configure form based security for REST APIs, and how we have done that in Spring Lemon.

UserDetails

Spring Security uses a UserDetails object to hold the data about a logged-in user. It should have a *username*, a *password* and a set of *authorities*. The authorities are used for determining the rights of the user.

Your users should be mapped to `UserDetails`. In Spring Lemon, we simply let AbstractUser implement `UserDetails`:

```
@MappedSuperclass
public abstract class AbstractUser
      <U extends AbstractUser<U,ID>, ID extends Serializable>
extends VersionedEntity<U, ID>
implements UserDetails {
```

This needs us to implement `getUsername`, `getPassword`, `getAuthorities` and a few more methods, which are discussed next.

getUsername

This should return the login id of the user. By default, we have decided to use *email* as the login id in Spring Lemon. So, our `getUsername` looks like this:

```
@Override
public String getUsername() {
      return email;
}
```

In addition to *getUsername*, AbstractUser also has a *setUsername*, and you need to override both these methods if you decide to use another field instead of `email` for logging in.

getPassword

This would be nothing but the getter of our *password* property:

```
@Override
public String getPassword() {
      return password;
}
```

getAuthorities

Spring Security calls `getAuthorities` to know the rights of the user. It should return a collection of GrantedAuthoritys.

In Spring Lemon, we build the collection from our `roles` field. It's done in the *decorate* method, and stored it in a transient field. That field is then returned from *getAuthorities*.

Below is the relevant code:

```
@Transient
protected Collection<GrantedAuthority> authorities;

protected void computeAuthorities() {

	authorities = roles.stream()
		.map(role -> new SimpleGrantedAuthority("ROLE_" + role))
		.collect(Collectors.toCollection(() ->
			new HashSet<GrantedAuthority>(roles.size() + 2)));

	if (goodUser) {

		authorities.add(new SimpleGrantedAuthority("ROLE_"
				+ LemonSecurityConfig.GOOD_USER));

		if (goodAdmin)
			authorities.add(new SimpleGrantedAuthority("ROLE_"
					+ LemonSecurityConfig.GOOD_ADMIN));
	}
}

public U decorate(U currentUser) {
	...
	computeAuthorities();
	...
}

@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
	return authorities;
}
```

Other methods

AbstractUser also needs to implement `isAccountNonExpired`, `isAccountNonLocked`, `isCredentialsNonExpired` and `isEnabled` in order to comply with *UserDetails*. They may not be useful commonly, and so AbstractUser provides minimal implementations for those. You can, of course, override those in your concrete user class.

Some utility methods

Before we proceed further, it would be useful to discuss about a few utility methods. These are defined in the LemonUtil class.

Getting the current user

Data about the current-user will be needed many times in your application. It would be available as the *principal* in the authentication object that Spring Security maintains per logged in user. Below are the couple of methods in Spring Lemon for retrieving it:

```
public static <U extends AbstractUser<U,ID>, ID extends Serializable>
      U getUser() {

      Authentication auth = SecurityContextHolder
            .getContext().getAuthentication();

      return getUser(auth);
}

public static <U extends AbstractUser<U,ID>, ID extends Serializable>
      U getUser(Authentication auth) {

    if (auth != null) {
      Object principal = auth.getPrincipal();
      if (principal instanceof AbstractUser<?,?>) {
        return (U) principal;
      }
    }
    return null;
}
```

You will need the first one most of the times. The second one would be useful when you already have got the *authentication* object.

Logging a user in

Normally we don't need to log a user in programmatically - Spring Security does that. Still, in some cases, e.g. after a new user signs up, we need this. So, we have the following method in Spring Lemon:

```
public static <U extends AbstractUser<U,ID>, ID extends Serializable>
void logIn(U user) {

    user.decorate(user); // decorate self
    Authentication authentication = // make the authentication object
      new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
    SecurityContextHolder.getContext().setAuthentication(authentication);
}
```

As you see above, the given user is first decorated, so that its authorities and other transient attributes are set. Then, an authentication object is made out of it and put in the security context.

Logging a user out

Sometimes we need to log a user out programmatically. Spring Lemon has the following utility method doing this:

```
public static void logOut() {
	SecurityContextHolder.getContext().setAuthentication(null);
}
```

UserDetailsService

Spring Security would need us to provide a UserDetailsService for fetching the user data from the database or wherever it is stored. In Spring Lemon, we have the UserDetailsServiceImpl service doing that job:

```
@Service
@ConditionalOnProperty(name="lemon.enabled.user-details-service",
	matchIfMissing=true)
public class UserDetailsServiceImpl
	<U extends AbstractUser<U,ID>, ID extends Serializable>
implements UserDetailsService {

	@Autowired
	protected AbstractUserRepository<U,ID> userRepository;

	@Override
	public UserDetails loadUserByUsername(String username)
			throws UsernameNotFoundException {

		U user = findUserByUsername(username)
			.orElseThrow(() -> new UsernameNotFoundException(username));

		user.decorate(user);
		return user;
	}

	protected Optional<U> findUserByUsername(String username) {
		return userRepository.findByEmail(username);
	}
}
```

Note the following:

1. The `@ConditionalOnProperty` class annotation will prevent creation of the above service if there would be

   ```
   lemon.enabled.user-details-service=false
   ```

   in application properties. Provide this property if you don't want to use the above service and code yours instead.

2. The overridden `loadUserByUsername` method receives the login id of the user, fetches the user and returns it. For fetching, it uses the `findUserByUsername` protected method. So, if you using some other field but not email as the login id, you can
    a. Create a subclass and override `findUserByUsername`.
    b. Make the subclass a *@Service*.
    c. Provide a property *lemon.enabled.user-details-service=false*, so that *UserDetailsServiceImpl* isn't made a service.

Configuring Spring Security

Now that we have a `UserDetails` and a `UserDetailsService`, let's discuss how to configure Spring Security. A simple way to do it is to have a configuration class extending WebSecurityConfigurerAdapter, and override some of its methods. Here is how it looks as a whole in Spring Lemon - but before looking at that, let's understand it in pieces.

At the minimum, it would look like this:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
       ...
    }

}
```

As you see above, we need to override the configure method for configuring various security parameters. To begin with, we can code it like this:

```
@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure[1](HttpSecurity http) throws Exception {
       http
              .formLogin[2]()
                     .and()
              .logout()
                     .invalidateHttpSession(true)
                     .deleteCookies("JSESSIONID")
                     .and()
              .authorizeRequests[3]()
                     .antMatchers("/**").permitAll();
    }
}
```

`configure`[1] tells Spring Security many things, as given below.

`formLogin`[2] sets up Spring to receive `POST /login` requests with `username` and `password` parameters, and log the user in if everything is found in order.

```
.logout()
      .invalidateHttpSession(true)
      .deleteCookies("JSESSIONID")
```

tells Spring to log the user out when `POST /logout` is received, and invalidate the session as well as delete JSESSIONID cookie after that.

What is `authorizeRequests`[3] ? Read on.

Authorization

Whereas *authentication* means logging the users in, *authorization* means checking the rights of the users before granting access to parts of the application.

Spring enables you to enforce authorization both at request level and method level.

Request level authorization

You can configure Spring Security to restrict access to the urls. `authorizeRequests`[3] is used for that purpose. For example,

```
            .authorizeRequests()
                  .mvcMatchers("/secure/**").authenticated()
                  .mvcMatchers("/**").permitAll();
```

will tell Spring to grant access to all the urls beginning with `/secure` only to logged in users, whereas all other urls can be accessed by everybody.

Method level authorization

Apart from restricting urls, you can also restrict access to your service methods. To enable this with *pre/post* annotations, annotate the *SecurityConfig* class with `@EnableGlobalMethodSecurity` as below:

```
@Configuration
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
```

For restricting access to a method, we can then annotate it with pre/post annotations like *@PreAuthorize*. See this example:

```
@PreAuthorize("isAuthenticated()")
public void verifyUser(String verificationCode) {
```

`@PreAuthotize` above takes a *Spring EL* expression, which is `isAuthenticated()` in the above case.

Here is a list of methods that can be used in the expression. Of those, `hasPermission(Object target, Object permission)` is an interesting one, which we discuss below.

hasPermission

hasPermission determines whether the currently logged-in user should be allowed a particular operation on a particular object. For example, look at below how a service method could be annotated:

```
@PreAuthorize("hasPermission(#user, 'edit')")
public void updateUser(U user, U updatedUser) {
    ...
}
```

The above method will be executed only if the current-user has *'edit'* permission to the *user* parameter. Otherwise, an `AccessDeniedException` will be thrown.

For this to work, you need to supply a PermissionEvaluator, as below:

```
@Component
public class PermissionEvaluatorImpl implements PermissionEvaluator {

      @Override
      public boolean hasPermission(Authentication auth,
                  Object user, Object permission) {

            // return true if auth has permission permission for the user.
            // Current-user can be obtained from auth.
      }

      ...
}
```

As you see above, the PermissionEvaluatorImpl has a `hasPermission` method, in which you do the actual checking. It receives three parameters:

1. The Spring Security authentication object – you can get the current-user data from this object
2. The object for which the permission has to be checked
3. The permission

Using these parameters, you should decide whether to allow the annotated method to be executed. Return `true` if yes, else return `false`.

In Spring Lemon, we have the LemonPermissionEvaluator doing this job, which looks as below:

```
@Component
public class LemonPermissionEvaluator
<U extends AbstractUser<U,ID>, ID extends Serializable>
implements PermissionEvaluator {

    @Override
```

```
    public boolean hasPermission(Authentication auth,
                    Object targetDomainObject, Object permission) {

      if (targetDomainObject == null) // if no domain object is provided,
            return true;              // let's pass, allowing the service method
                                      // to throw a more sensible error message

      // Let's delegate to the entity's hasPermission method
      VersionedEntity<U, ID> entity = (VersionedEntity<U, ID>) targetDomainObject;
      return entity.hasPermission(LemonUtil.getUser(auth), (String) permission);
    }

    @Override
    public boolean hasPermission(Authentication authentication,
            Serializable targetId, String targetType, Object permission) {

      throw new UnsupportedOperationException(
            "hasPermission() by ID is not supported");
    }
}
```

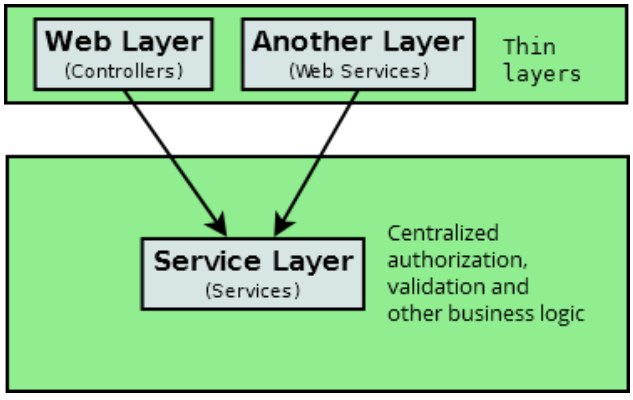
As you see, it actually uses the hasPermission method of the entity class, thus giving us a clean pattern. In summary, to restrict your entities based on the currently logged-in user,

1. Annotate the service method with `@PreAuthorize("hasPermission….`
2. Provide a `hasPermission` method in your entity class.

There is much more to Spring Security's method level authorization. You can find about it in the Spring Security reference material.

Request vs method level authorization

So, should you restrict URLs, or use method level authorization? In Spring Lemon, we preferred to use method-level authorization. This way, our authorization rules would stay centralized in the service layer. Thus, we can add other layers without bothering about the authorization rules. Remember the picture below?



Preventing redirection after login/logout etc.

By default, when a user successfully logs in, Spring Security redirects him to the home page. When a login fails, or after a successful logout, the user is redirected back to the login page. Also, on trying to access URLs for which a user does not have sufficient rights, he is redirected to the login page.

These redirections look great in traditional multi-page applications. But, in REST APIs or single page applications, your requests goes programmatically, e.g. through AJAX, and so redirections won't be desirable. Instead, the API should send a success (200) response along with the user data, or a unauthorized (40x) response. At the client side, you would receive this data and update the UI accordingly.

This behavior can be obtained by providing our own login/logout/exception handlers in `configure`[1]:

```
    @Override
    protected void configure[1](HttpSecurity http) throws Exception {
       http
              .formLogin()
                     .successHandler(your authentication success handler object)
                     .failureHandler(your authentication failure handler object)
                     .and()
              .logout()
                     .logoutSuccessHandler(your logout success handler object)
                     ...
                     .and()
              .exceptionHandling()
                     .authenticationEntryPoint(new Http403ForbiddenEntryPoint())
              ...
    }
```

Let's now talk about each of the handlers above.

Authentication Success Handler

As you see above, in

```
                     .successHandler(your authentication success handler object)
```

we need to provide an object of our own authentication success handler class, which can look like this:

```
@Component
public class AuthenticationSuccessHandler
      extends SimpleUrlAuthenticationSuccessHandler {

    @Autowired
    private ObjectMapper objectMapper;

    @Autowired
```

```
    private LemonService<?,?> lemonService;

    @Override
    public void onAuthenticationSuccess(HttpServletRequest request,
            HttpServletResponse response,
           Authentication authentication)
    throws IOException, ServletException {

      [1]response.setStatus(HttpServletResponse.SC_OK);
      [2]response.setContentType(MediaType.APPLICATION_JSON_VALUE);

      [3]AbstractUser<?,?> currentUser = lemonService.userForClient();

      [4]response.getOutputStream().print(
                  objectMapper.writeValueAsString(currentUser));

      [5]clearAuthenticationAttributes(request);
    }
}
```

It would sends a 200 OK response with the data about the currently logged in user. Specifically,

- ✓ [1] sets the response status as 200
- ✓ [2] sets the content type to `application/json`.
- ✓ [3] fetches the current-user data, using userForClient() of *LemonService*, which we will discuss later.
- ✓ [4] writes the JSON value of current-user to the response stream. Notice that we are using an injected `objectMapper` for converting our current-user to JSON. This `objectMapper` was created by Spring MVC for converting return data from request handlers to JSON.
- ✓ [5] is just copied from the base SimpleUrlAuthenticationSuccessHandler class. Look at that if you want to understand its use.

Complete source of Spring Lemon's authentication success handler can be viewed here.

Authentication Failure Handler

The following line in *configure*[2] configures our own failure handler:

```
                .failureHandler(your authentication failure handler object)
```

In fact, we won't have to code a class for it. The SimpleUrlAuthenticationFailureHandler provided by Spring, if instantiated without any arguments, would fit our purpose perfectly.

Logout Success Handler

This could be as simple as below:

```
public class LemonLogoutSuccessHandler
    implements LogoutSuccessHandler {
```

```
    @Override
    public void onLogoutSuccess(HttpServletRequest request,
            HttpServletResponse response, Authentication authentication)
            throws IOException, ServletException {

        response.setStatus(HttpServletResponse.SC_OK);
    }
}
```

Here is the actual class in Spring Lemon.

Authentication Entry Point

When a user with insufficient rights tries to access a restricted URL, he is redirected to the login page by default. But, by coding

```
            .exceptionHandling()
                  .authenticationEntryPoint(new Http403ForbiddenEntryPoint())
```

in *configure*[1] as we saw above, we can tell Spring to use the use its Http403ForbiddenEntryPoint class to generate the response, which sends a *403 Forbidden* response.

Remember Me

You must have seen the *Remember Me* check box on almost every login form found on the web. Clicking on it, the user is remembered by the browser across sessions.

To have it in your application, you will need to supplement your login requests with a remember-me parameter, like this:

```
POST /login
username=someLoginId&password=somePassword&rememberMe=true
```

Let's see what configuration we need for this in the server side. Spring Security has very good support for it. You first need to create an instance of Spring's RememberMeServices, like this:

```
    @Autowired
    private UserDetailsService userDetailsService;

    @Bean
    public RememberMeServices rememberMeServices() {

        TokenBasedRememberMeServices rememberMeServices =
            new TokenBasedRememberMeServices(rememberMeKey[1], userDetailsService[2]);
        [3]rememberMeServices.setParameter("rememberMe");
        [4]rememberMeServices.setCookieName("rememberMe");
        return rememberMeServices;

    }
```

Here we have used Spring's *TokenBasedRememberMeServices*, which would suffice for most projects. But, you can also use Spring's *PersistentTokenBasedRememberMeServices*, or code your own class.

TokenBasedRememberMeServices uses a cookie to remember the user. As you see above, when creating an instance of it, we need to supply a `rememberMeKey`[1], which is a string for encrypting the cookies. It should be a secret, kept in the application properties. We also need to supply our `userDetailsService`[2].

Line [3] sets the name of the request parameter, and line [4] sets the cookie name. Spring provides some defaults if you don't set these, but those defaults may change in new versions of Spring. So, it's safer to provide our own names.

After you have configured a `rememberMeServices` bean as above, you need to wire it in the `configure(HttpSecurity http)` method:

```
      http
            ...
            .rememberMe()
                  .key(rememberMeKey)
                  .rememberMeServices(rememberMeServices())
            ...
```

That's all you need to support the famous "Remember Me" feature! See the Spring Security reference material to know more about it.

Switching User

Say you are a support guy for an application. You hear a complaint from a user that he is unable to use feature X. But you find nothing wrong with it. What would you do?

Probably, you would like to *login as that user* and try to face the issue yourself. But, it wouldn't be decent to ask the user to share his password.

Spring Security provides a SwitchUserFilter for such needs. After configuring it, you can switch to another user and then switch back, as described here.

To have this feature, you first need to create an instance of *SwitchUserFilter*, as below:

```
@Bean
public SwitchUserFilter switchUserFilter() {
      SwitchUserFilter filter = new SwitchUserFilter();
      filter.setUserDetailsService(userDetailsService);
      filter.setSuccessHandler(authenticationSuccessHandler);
      filter.setFailureHandler(authenticationFailureHandler);
      return filter;
}
```

Then, in the `configure(HttpSecurity http)` method, you can add the filter after Spring's FilterSecurityInterceptor, as below:

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http
        ...
        .addFilterAfter(switchUserFilter(), FilterSecurityInterceptor.class);
```

With this configured, now switching can be done by using

```
POST /login/impersonate
username=user1@example.com
```

and switching back can be done by using

```
POST /logout/impersonate
```

Note that it's your job to ensure that existing user must have enough rights for the switch. The common practice is to restrict `/login/impersonate` only to ADMINs, and and `/logout/impersonate` to authenticated users, like this:

```
                    .authorizeRequests()
                          .antMatchers("/login/impersonate*").hasRole("ADMIN")
                          .antMatchers("/logout/impersonate*").authenticated()
                          .antMatchers("/**").permitAll();
```

SwitchUserFilter supports multi-level switching.

Encrypting Password

We shouldn't store user passwords as plaintext in database. Instead, we should do the following:

1. When a user signs up or changes password, encrypt the password before storing it.
2. When someone signs in, match the given password against the encrypted one (using some algorithm).

For this, we need a PasswordEncoder object, which we can have like this:

```
@Bean
public PasswordEncoder passwordEncoder() {
  return new BCryptPasswordEncoder();
}
```

It uses the *BCryptPasswordEncoder* provided by Spring. But, if you like, you can roll out your own, or use some other one provided by Spring.

In Spring Lemon, it's coded in the LemonSecurityConfig class.

When signing up, we can then encode the password this way:

```
user.setPassword(passwordEncoder.encode(password));
```

When someone logs in, Spring Security's authentication manager, on finding a PasswordEncoder bean in the application context, would use that to match the given password against the encrypted one in the database. Internally, it uses `passwordEncoder`'s `matches` method, like this:

```
if (passwordEncoder.matches(plainPassword, encryptedPassword))
      ...
```

CSRF

CSRF is a well-known security vulnerability. If you do not know about it, you may like to get familiar with it before we proceed. Spring Security reference material explains it beautifully.

Spring Security has great support for preventing CSRF. In summary, it generates a CSRF token which your client application will have to know and send back along with PATCH, POST, PUT, or DELETE requests, either as a request parameter or as a header. The sent token is then matched against the generated one, and an exception is thrown if they do not match.

How does the client side know the token?

Spring Security has a CsrfFilter, which puts the token in a request parameter. That means, while rendering a traditional JSP Form, you can include the token like this:

```
<form action="/foo/5/update" method="post">
      <input type="text" ... />
      <input type="submit" value="Update" />
      <input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
</form>
```

This way, the token reaches the client, which gets re-submitted along with the form.

But, this won't work in single page applications, where your forms are pure HTML. In such case, you can configure a CookieCsrfTokenRepository to tell Spring Security to store the token in a cookie. It can be configured in our `configure(HttpSecurity http)` method, as below:

```
@Override
protected void configure(HttpSecurity http) throws Exception {

    http
      ...
      .csrf()

          .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse());
      ...
}
```

At the client side, before doing any POST (or PUT or PATCH or DELETE), we must first send some GET request which fetches the cookie for us. Then, when doing the POST, we send back the token as a header. For example, if you are using *AngularJS*, you can write an `$http` interceptor like this:

```
angular.module('myApp')
  .factory('XSRFInterceptor', function ($cookies, $log) {

    var xsrfToken;

    var XSRFInterceptor = {

      request: function(config) {

        if (xsrfToken) {
          config.headers['X-XSRF-TOKEN'] = xsrfToken;
          $log.info("X-XSRF-TOKEN being sent to server: " + xsrfToken);
        }

        return config;
      },

      response: function(response) {

        var newToken = $cookies.get('XSRF-TOKEN');

        if (newToken) {
          xsrfToken = newToken;
          $log.info("XSRF-TOKEN received from server: " + xsrfToken);
        }

        return response;
      }
    };

    return XSRFInterceptor;
  });
```

> You don't actually have to code the above interceptor in AngularJS unless you are doing cross domain requests. AngularJS automatically sends back our XSRF-TOKEN cookie as X-XSRF-TOKEN header.

Finally, be aware that the cookie gets removed or changed after certain events, like logout. Hence, after these events, your client application will again need to send some GET request before they can send POST kind of requests. Supporting a simple `GET /ping` kind of controller method would be handy for this. In fact, we have one such method in our *LemonController*, which we haven't discussed yet.

LemonSecurityConfig

All that we discussed above is assembled in the LemonSecurityConfig class in Spring Lemon. Have a look at that class.

Notice that it is an abstract class, with well granulated and overridable member functions. In your applications, you need define a `@Configuration` class extending `LemonSecurityConfig`. This is a simple example:

```
@Configuration
public class MySecurityConfig extends LemonSecurityConfig {

}
```

Below is another example, to add a restriction for urls beginning with `/admin`:

```
@Configuration
public class MySecurityConfig extends LemonSecurityConfig {

	@Override
	protected void authorizeRequests(HttpSecurity http) throws Exception {
		http.authorizeRequests()
			.mvcMatchers("/admin/**").hasRole("ADMIN");
		super.authorizeRequests(http);
	}
}
```

CORS

When developing a single page application, your client code interacts with the server code purely using API calls. So, they could very well be two separate projects, developed and hosted separately. In fact, we like having those separate, because of modularity, cleanliness and separation of concerns. For example, your Spring application could be hosted at Pivotal Could Foundry, available via `example.cfapps.io`, whereas your client application could be hosted with some static website host, available to users at `www.example.com`.

Looks great. But there is one problem. Due to the same origin policy, before allowing JavaScript at `www.example.com` to call your API at a different domain, browsers will first check whether your API allows it. It's called the CORS mechanism.

Your API then should respond with a few headers, as below:

1. *Access-Control-Allow-Origin*: The website that's allowed to access the API. E.g. `https://www.example.com`, assuming your front-end is hosted there. If you want to allow multiple websites to access the API, you need to set this header dynamically. We will see in a moment how to do that.
2. *Access-Control-Allow-Methods*: Comma separated methods those should be allowed. E.g. `GET,HEAD,POST, PUT,DELETE,TRACE,OPTIONS,PATCH`

3. *Access-Control-Allow-Headers*: Custom request headers that should be allowed. E.g. `x-requested-with,x-xsrf-token`.
4. *Access-Control-Expose-Headers*: Custom response headers that the browser should be permitted to read. E.g. `x-xsrf-token`.
5. *Access-Control-Allow-Credentials*: This should be set to `true`, as we want to support logging in.
6. *Access-Control-Max-Age*: No of seconds the results of a preflight request can be cached, e.g. `3600`.

You might never need to dig into more details on CORS – but if you are interested, there is ample of material on the Internet.

Spring Boot has some built-in support for CORS. But, as of this writing, it does not seem to support many use cases. So, a common practice now is to return the above headers by coding a filter. In Spring Lemon, we have the LemonCorsFilter for this, which looks as below:

```
@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
@ConditionalOnProperty(name="lemon.cors.allowedOrigins")
public class LemonCorsFilter extends OncePerRequestFilter {

	@Autowired
	protected LemonProperties properties;

	@Override
	protected void doFilterInternal(HttpServletRequest request,
			HttpServletResponse response, FilterChain filterChain)
			throws ServletException, IOException {

		Cors cors = properties.getCors();

		// origin as provided by the browser
		String origin = request.getHeader("Origin");

		response.setHeader("Access-Control-Allow-Origin",
			ArrayUtils.contains(cors.getAllowedOrigins(), origin)
			origin : properties.getApplicationUrl());

		response.setHeader("Access-Control-Allow-Methods",
			StringUtils.join(cors.getAllowedMethods(), ","));

		response.setHeader("Access-Control-Allow-Headers",
			StringUtils.join(cors.getAllowedHeaders(), ","));

		response.setHeader("Access-Control-Expose-Headers",
				StringUtils.join(cors.getExposedHeaders(), ","));

		response.setHeader("Access-Control-Max-Age",
			Long.toString(cors.getMaxAge()));

		response.setHeader("Access-Control-Allow-Credentials", "true");
```

```
            if (!request.getMethod().equals("OPTIONS"))
                  filterChain.doFilter(request, response);
      }
}
```

Note that

1. We have annotated the class with `@Order(Ordered.HIGHEST_PRECEDENCE)`, because preflight OPTIONS requests should hit this first.
2. The property `lemon.cors.allowed-origins` holds the list of websites that should be permitted to access the API. For example, if you want your test deployment to be accessed both from *testsvr.com* and *localhost:9000*, have your *application-test.properties* include:

   ```
   lemon.cors.allowed-origins: http://testsvr.com,http://localhost:9000
   ```

   If you don't provide this property, CORS support would be disabled – because of the `@ConditionalOnProperty` class annotation.

3. Browser would send the front-end website name in an `Origin` header. Our code checks if it is whitelisted in *lemon.cors.allowed-origins*. If it is, we send it back in the *Access-Control-Allow-Origin* header, thus permitting it. If it's not, we just send something else (the *application-url*), thus blocking the request.
4. OPTIONS requests, which would be pre-flight requests, won't be needed to pass down the filter chain. Other requests are passed down the chain.

When using Spring Lemon, if the above filter does not satisfy your need, disable it by not providing the *lemon.cors.allowed-origins* property, and coding your own mechanism instead.

JSON Vulnerability

If your API is going to be used from a browser based application, you may like to address the JSON vulnerability. Precisely, your API can prefix the JSON responses with something like `)]}',\n`, and the client application can strip it out. Let's see how.

Prefixing JSON responses

For converting return values to JSON, Spring Boot automatically configures a `MappingJackson2HttpMessageConverter`, unless we provide one. But, that converter does not do the prefixing. Hence, we can provide our own converter, just by configuring a bean like this:

```
@Bean
public MappingJackson2HttpMessageConverter
mappingJackson2HttpMessageConverter() {

    MappingJackson2HttpMessageConverter converter =
            new MappingJackson2HttpMessageConverter();
```

```
    converter.setJsonPrefix(")]}',\n");

    return converter;
}
```

Spring would then use this one.

In Spring Lemon, it's coded in the LemonConfig class, and is annotated with

```
@ConditionalOnProperty(name="lemon.enabled.json-prefix", matchIfMissing=true)
```

This tells Spring not to create the bean if you have a property:

```
lemon.enabled.json-prefix: false
```

So, when using Spring Lemon in your applications, if you don't want your JSON responses to be prefixed, have the above line in some application properties.

Removing the prefix

Before using the response, the client will need to remove the prefix. The $http service of AngularJS does it automatically. Look at your client library for whether it is done automatically or how to do it. For example, when testing Spring Lemon with *RestAssured*, we use this filter.

11 Sending mails

Almost all applications would need sending mails. For example, when a user signs up, a mail could be sent to him for verifying his email id. When a user forgets his password, a secret token could be mailed to him to enable him to reset his password.

In this chapter, we will see how to send SMTP mails from a Spring Boot application. We will also see how to mock sending mails in development or test environment, and write the content to log instead. We will also see how to put the code in a common library such as spring Lemon, and allow the application developer to replace our mail component, if needed.

Sending SMTP Mails

For sending SMTP mails, Spring Framework comes with a JavaMailSender.

Configuring JavaMailSender

In a Spring boot application, it's auto-configured if suitable dependency and properties are discovered. Specifically, you need to

1. Add the following dependency to pom.xml, assuming you are using maven:

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

2. Add relevant properties to application.properties. For example, the following properties would be needed for using Gmail service:

```
spring.mail.host = smtp.gmail.com
spring.mail.username = xxxxxx@gmail.com
spring.mail.password = xxxxxx

spring.mail.properties.mail.smtp.auth = true
spring.mail.properties.mail.smtp.socketFactory.port = 465
spring.mail.properties.mail.smtp.socketFactory.class = javax.net.ssl.SSLSocketFactory
spring.mail.properties.mail.smtp.socketFactory.fallback = false
spring.mail.properties.mail.smtp.ssl.enable = true
```

   Note that you should have activated Google 2-step verification, and the password should be an application password instead of your regular password.

Sending mails

There are multiple ways to send mails using a JavaMailSender. We prefer using a MimeMessageHelper:

```
MimeMessage message = javaMailSender.createMimeMessage();
MimeMessageHelper helper;

helper = new MimeMessageHelper(message, true); // true indicates multipart message
helper.setSubject(subject);
helper.setTo(to);
helper.setText(body, true); // true indicates html

// continue using the helper for more functionality, like adding attachments etc.

javaMailSender.send(message);
```

Structuring the code

You may not like to send real SMTP mails from the development environment or while doing automated testing. So, in our application, we shouldn't use the above concrete code. Instead, using an interface could be a good idea.

MailSender interface

The interface can be coded like this:

```
public interface MailSender {

	void send(String to, String subject, String body);
}
```

We can then code a couple of its implementations, say *MockMailSender* and *SmtpMailSender*. The former would write the mail content just to the log, while the later would actually send mails.

MockMailSender

MockMailSender can be coded like this:

```
public class MockMailSender implements MailSender {

	private static final Log log = LogFactory.getLog(MockMailSender.class);

	@Override
	public void send(String to, String subject, String body) {
		log.info("Sending mail to " + to);
		log.info("Subject: " + subject);
		log.info("Body: " + body);
	}
```

```
}
```

SmtpMailSender

SmtpMailSender can be coded like this:

```
public class SmtpMailSender implements MailSender {

	private JavaMailSender javaMailSender;

	public void setJavaMailSender(JavaMailSender javaMailSender) {
		this.javaMailSender = javaMailSender;
	}

	@Override
	@Async
	public void send(String to, String subject, String body)
			throws MessagingException {

		MimeMessage message = javaMailSender.createMimeMessage();
		MimeMessageHelper helper;

		helper = new MimeMessageHelper(message, true);
		helper.setSubject(subject);
		helper.setTo(to);
		helper.setText(body, true);

		javaMailSender.send(message);
	}
}
```

Note that we are using `@Async` for sending the the mail asynchronously, so that the calling request can be completed without waiting for the mail to be sent. For this to work, we should have annotated some configuration class with `@EnableAsync`. Spring Lemon has it on the LemonConfig class.

Configuring MailSender

So, a mail sender can now be configured by using a configuration class, like this:

```
@Configuration
public class MailConfiguration {

	@Bean
	@ConditionalOnProperty(name="spring.mail.host", matchIfMissing=true)
	public MailSender mockMailSender() {
		return new MockMailSender();
	}

	@Bean
	@ConditionalOnProperty("spring.mail.host")
```

```
        public MailSender smtpMailSender(JavaMailSender javaMailSender) {

                SmtpMailSender mailSender = new SmtpMailSender();
                mailSender.setJavaMailSender(javaMailSender);
                return mailSender;
        }
}
```

Note the use of `@ConditionalOnProperty` to configure a *MockMailSender* when a *spring.mail.host* property is not available. So, to use MockMailSender instead of the SmtpMailSender, just don't provide the *spring.mail.host* property.

> How to provide properties based on environments is discussed earlier.

Allowing developers to provide their implementations

If *MockMailSender* or *SmtpMailSender* would not suit the requirements, an application developer should be able to code and use his own implementation. So, our configuration class can be annotated with:

```
@Configuration
@ConditionalOnMissingBean(MailSender.class)
public class MailConfiguration {
```

This configuration will then be done only if there is no other *MailSender* configured.

> @ConditionalOnMissingBean seem like a great feature, but it is to be used with caution. See my stackoverflow answer for more details.

12 Coding the API

Until this chapter, we have been coding only operational code, e.g. configuration, helper and utility classes. This chapter onwards, we will learn how to actually develop an elegant API, by using whatever we have coded so far. The best way to learn it would be to examine how the Spring Lemon API is written. That also would make you a master of Spring Lemon, and you can reuse it fluently in your projects.

Spring MVC REST support

Let's first have a quick a recap of how Spring MVC supports developing REST APIs.

Using Spring MVC, we can write *request handler methods* in *controller classes*, and map those to incoming requests.

A traditional request handler method that uses some view technology like *JSP* or *Thymeleaf* may look like this:

```
@Controller
public class SomeController {

	@GetMapping("/signup")
	public String signup(Model model) {

		...

		return "signup";

	}
}
```

The return value helps identifying a view, e.g. a JSP, and the request is then forwarded to that view, for rendering HTML.

But, when coding an API, we would not want to render HTML. Instead, we would want to send data in JSON or some other format like XML. To do so, you need to annotate either the class with `@RestController`, or the method with `@ResponseBody`. These annotations tell Spring to convert the return value to the required format, e.g. JSON. So, this also needs us to configure some *converter* for doing the conversion. When using Spring Boot, a JSON converter is configured by default.

In summary, using Spring Boot, the common practice for coding a JSON API is to annotate the controllers with `@RestController` and let the handler methods return Java objects directly, like this:

```
@RestController
public class SomeController {
```

```
	@GetMapping("/some-path")
	public SomeObject getContext() {

		...

		return someObject;
	}
}
```

Abstract controllers and services

In Spring Lemon, we have an abstract controller and an abstract service looking like this:

```
public abstract class LemonController
	<U extends AbstractUser<U,ID>, ID extends Serializable> {
	...
}
```

```
@Validated
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public abstract class LemonService
	<U extends AbstractUser<U,ID>, ID extends Serializable> {
	...
}
```

The first one, `LemonController`, holds the common handler methods like *signup* and *change profile*. These methods are very sleek – they don't contain any business logic. The second one, `LemonService`, holds methods containing the actual business logic, which are called from the controller. It's composed of small decoupled methods, which can be selectively overridden by your application.

In your application, you would extend these classes as below:

```
@RestController
@RequestMapping("/api/core")
public class MyController extends LemonController<User, Long> {

}
```

```
@Service
public class MyService extends LemonService<User, Long> {

}
```

As you see, `MyController` would make the Spring Lemon API available at `/api/core/`.

With this much knowledge, next chapter onwards, we are going to analyze each function of Spring Lemon API, one-by-one. LemonController and LemonService would be referred all the time, so keep Spring Lemon source code open in your IDE side-by-side.

13 Ping

While discussing about CSRF earlier, we felt the need of a *ping* request to fetch the CSRF token. Coding a *ping* function would be simple:

```
public abstract class LemonController
      <U extends AbstractUser<U,ID>, ID extends Serializable> {

      @GetMapping("/ping")
      public void ping() {
            // no need to do anything
      }
}
```

14 Get Context Data

Certain of your application properties, e.g. the *reCAPTCHA public key*, would be needed by the client applications of your API. The client application will typically fetch these at startup. Clients will also need the *current-user data* at startup, so that customized screens and menus can be rendered.

Spring Lemon provides a *get context* function for this.

```
@Autowired
private LemonService<U, ID> lemonService;

@GetMapping("/context")
public Map<String, Object> getContext() {

	Map<String, Object> context =
		LemonUtil.mapOf("context", lemonService.getContext(),
				"user", lemonService.userForClient());

	return context;
}
```

As you see, it returns a map with a couple of entries:

1. Application properties needed at the client side
2. Current-user data

The service methods `getContext()` and `userForClient()`, discussed below, do the actual job of building the data.

getContext()

`getContext()` service method looks like this:

```
@Autowired
private LemonProperties lemonProperties;

public Map<String, Object> getContext() {

	Map<String, Object> context = new HashMap<String, Object>(2);
	context.put("reCaptchaSiteKey", properties.getRecaptcha().getSitekey());
	context.put("shared", properties.getShared());
	return context;
}
```

We are returning a map containing *reCaptchaSiteKey* and *shared*. If you recall, *shared* is a map of properties prefixed with `lemon.shared`, like this:

```
#
## Properties to be passed to client
## should be prefixed with "lemon.shared"
#
lemon.shared.fooBar: 123
```

So, if you want to send some properties to the client, simply have those prefixed with `lemon.shared`.

An application developer can override this method if needed.

userForClient()

`userForClient()` looks like this:

```
public U userForClient() {
	return userForClient(LemonUtil.getUser());
}

protected U userForClient(U currentUser) {

	if (currentUser == null)
		return null;

	U user = newUser();
	user.setIdForClient(currentUser.getId());
	user.setUsername(currentUser.getUsername());
	user.setRoles(currentUser.getRoles());
	user.decorate(currentUser);

	return user;
}

abstract protected U newUser();
```

The first method is the one called from the controller. It calls the second one, passing the current-user. The second one

1. Creates a new user by calling the abstract `newUser()` method. Your applications should override `newUser()` and return an instance of your concrete user class. See here for an example.
2. Fills the new user with id, username and roles of the current-user. Remember that the username is actually email by default.
3. Decorate the new user, thus setting its transient flags.

To customize the response, you can override the second method. See here for an example.

15 Sign up

Spring Lemon's *signup* function signs a user up.

The controller method

Let's begin with analyzing the controller method:

```
@PostMapping("/users")
@ResponseStatus(HttpStatus.CREATED)
public U signup(@RequestBody @JsonView(SignupInput.class) U user) {

	lemonService.signup(user);
	return lemonService.userForClient();
}
```

As you see, this method

1. Is annotated with `@ResponseStatus(HttpStatus.CREATED)`, so it will let it return a *201 Created* response.
2. Receives the signup data as JSON, in the body of the request. Hence the `@RequestBody` annotation. The data would then be deserialized to a *User* object by the auto-configured JSON converter.
3. The `@JsonView(SignupInput.class)` would ensure that only those fields of the User class that are annotated with *@JsonView(SignupInput.class)* would be deserialized. This is an important measure to prevent malicious users from injecting extra fields into the User object.
4. Calls the `signup` service method, which will not only *sign the user up*, but also *log him in*.
5. Retuns the current login data, i.e. the user who was just signed up and logged in. You may ask why we don't return the current login data from the service method itself, thus wrapping up this controller method in a single line, like this:

   ```
   return lemonService.signup(user);
   ```

   That does not become possible because, as you will see next, we log the user in only after the database commit takes place, which happens after the service method returns.

Service methods

The signup service method looks like this:

```
@PreAuthorize("isAnonymous()")
@Validated(SignUpValidation.class)
```

```
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void signup(@Valid U user) {

	initUser(user);
	userRepository.save(user);

	LemonUtil.afterCommit(() -> {

		LemonUtil.logInUser(user);
		sendVerificationMail(user);
	});
}
```

Note the following:

1. Only new users should sign up. So, no point in allowing an already logged in user to sign up. The `@PreAuthorize` annotation, as discussed earlier, ensures that. Spring Security would throw an `AccessDeniedException` when a logged in user would try to access this service. That would be caught in our controller advice, and a 403 response would be sent to the client, as you know.
2. The @Validated annotation ensures that only those fields that have `SignUpValidation.class` as a validation group, like below, will be validated.

   ```
   @JsonView(SignupInput.class)
   @UniqueEmail(groups = {SignUpValidation.class})
   @Column(nullable = false, length = EMAIL_MAX)
   protected String email;
   ```

   So, if you add a new field to any concrete User class, remember to provide this validation group if the field is to be used while signing up.

3. `initUser`, which we will discuss next, is called to do the necessary data processing before the user is saved.
4. `userRepository.save` is called to save the user. But, remember that the transaction will be committed only after the method completes.
5. If the commit goes successful,
   a. The user is loged in.
   b. A verification mail is sent.

initUser

As you see above, `signup` has delegated the data processing to `initUser`. It is coded like this:

```
protected void initUser(U user) {

	user.setPassword(passwordEncoder.encode(user.getPassword()));
	makeUnverified(user);
}
```

Note that

1. It's a protected method, meant for overriding in the subclass if needed.
2. Password is encoded using the same encoder that is used by Spring Security.
3. `makeUnverified` is called to set the user as unverified.

makeUnverified

`makeUnverified` looks like this:

```
protected void makeUnverified(U user) {
      user.getRoles().add(Role.UNVERIFIED);
      user.setVerificationCode(UUID.randomUUID().toString());
}
```

As you see,

1. adds the UNVERIFIED role to the user
2. sets a random verification code, which would be useful while verifying the user.

sendVerificationMail

It sends the verification mail to the user:

```
protected void sendVerificationMail(final U user) {

    try {

      String verifyLink = properties.getApplicationUrl()
            + "/users/" + user.getVerificationCode() + "/verify";

      mailSender.send(user.getEmail(),
            LemonUtil.getMessage("com.naturalprogrammer.spring.verifySubject"),
            LemonUtil.getMessage(
                  "com.naturalprogrammer.spring.verifyEmail",  verifyLink));
      } catch (MessagingException e) {

            log.error(ExceptionUtils.getStackTrace(e));
      }

}
```

Note that

1. The verification link needs the url of the client web application. That should be provided in the application properties, like `lemon.application-url: https://example.com`. The default value, if it's not provided, would be http://localhost:9000, as you see in LemonProperties.java.
2. `LemonUtil.getMessage` fetches the internationalized messages corresponding to the given keys. The second one takes the verification link as a parameter. These should be defined in messages.properties (and all messages_*.properties), as in here.

3. If some exception occurs when sending the mail, we just log the error. We will provide a function for resending the verification mail, which the user can use in this case.

16 Resend verification mail

Users should be able to request for resending the verification mail if they miss the one that was sent while they signed up. Spring Lemon's *resend verification mail* function provides that functionality. Let's see how it's coded.

The controller method

It's coded like this:

```
@GetMapping("/users/{id}/resend-verification-mail")
public void resendVerificationMail(@PathVariable("id") U user) {

	lemonService.resendVerificationMail(user);
}
```

See how the path parameter `id` is automatically converted to a `User` parameter. Spring MVC does it, by using a *DomainClassConverter*, which is auto-configured as we have annotated one of our configuration classes – LemonConfig – with @EnableSpringDataWebSupport.

As you see, this method calls the `resendVerificationMail` service method, which we discuss next.

Service method

`resendVerificationMail` can be coded like this:

```
@PreAuthorize("hasPermission(#user, 'edit')")
public void resendVerificationMail(U user) {

	// The user must exist
	LemonUtil.check("id", user != null,
			"com.naturalprogrammer.spring.userNotFound").go();

	// must be unverified
	LemonUtil.check(user.getRoles().contains(Role.UNVERIFIED),
			"com.naturalprogrammer.spring.alreadyVerified").go();

	// send the verification mail
	sendVerificationMail(user);
}
```

Note that:

1. In `@PreAuthorize`, we are using hasPermission, which will invoke the permission evaluator in LemonPermissionEvaluator. That will then call the hasPermission method in our AbstractUser class. If you want it overriden, do so in your concrete User class.
2. We use LemonUtil.check to ensure that a user with such an id exists, and is indeed unverified.
3. For sending the mail, we are using the same sendVerificationMail that we discussed earlier.
4. We are not generating a new verification code. It would spare the user of getting confused which one was the latest verification mail, if he manages to get more than one.

We are going to use `@PreAuthorize("hasPermission(#user, 'edit')")` multiple times – whenever a service method needs *edit* permission on the given user. So, it'll actually be wise to define a meta annotation as below:

```
package com.naturalprogrammer.spring.lemon.permissions;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

import org.springframework.security.access.prepost.PreAuthorize;

@Retention(RetentionPolicy.RUNTIME)
@PreAuthorize("hasPermission(#user, 'edit')")
public @interface UserEditPermission {}
```

Spring Lemon has this in a `com.naturalprogrammer.spring.lemon.permissions` package.

Now, you can alter our *resendVerificationMail* service method as below:

```
@PreAuthorize("hasPermission(#user, 'edit')")
@UserEditPermission
public void resendVerificationMail(U user) {

    ...
```

17 Verify user

When a signed up user clicks on the verification link, he is taken to the front end application. He would then click a *verify* button there, which would send our API a request to verify his email. The detailed process is described in the API documentation. Let's now see how it is coded.

The controller method

It looks like this:

```
@PostMapping("/users/{verificationCode}/verify")
public U verifyUser(@PathVariable String verificationCode) {

	lemonService.verifyUser(verificationCode);
	return lemonService.userForClient();
}
```

This method calls the `verifyUser` service method, which we discuss next. It then returns the current-user data, which would enable the client to re-render its UI, as the user should now be a verified one.

Service method

It looks like this:

```
@PreAuthorize("isAuthenticated()")
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void verifyUser(@Valid @NotBlank String verificationCode) {

	// get the current-user from the session
	U currentUser = LemonUtil.getUser();

	// fetch a fresh copy from the database
	U user = userRepository.findOne(currentUser.getId());

	// ensure that he is unverified
	LemonUtil.check(user.getRoles().contains(Role.UNVERIFIED),
			"com.naturalprogrammer.spring.alreadyVerified").go();

	// ensure that the verification code of the user matches with the given one
	LemonUtil.check(verificationCode.equals(user.getVerificationCode()),
			"com.naturalprogrammer.spring.wrong.verificationCode").go();

	makeVerified(user); // make him verified
	userRepository.save(user);
```

```
        // after successful commit,
        LemonUtil.afterCommit(() -> {

                // Re-login the user, so that the UNVERIFIED role is removed
                LemonUtil.logIn(user);
        });
}
```

As you see above, we fetch a fresh copy of the current-user from the database, ensure that it's unverified, and then make it verified using the makeVerified protected method, which is discussed next.

After commit, we re-login the user so that Spring Security will know about the updated roles.

makeVerified

The `makeVerified` method would remove the *UNVERIFIED* role of the user, and reset the *verificationCode*:

```
protected void makeVerified(U user) {
        user.getRoles().remove(Role.UNVERIFIED);
        user.setVerificationCode(null);
}
```

18 Forgot password

See the API documentation to know how this function works. Below we discuss how it's coded.

The controller method

It's a single liner, calling the service function:

```
@PostMapping("/forgot-password")
public void forgotPassword(@RequestParam String email) {

	lemonService.forgotPassword(email);
}
```

Service methods

`forgotPassword` service method look like this:

```
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void forgotPassword(@Valid @Email @NotBlank String email) {

	U user = userRepository.findByEmail(email)
			.orElseThrow(() -> MultiErrorException.of(
				"com.naturalprogrammer.spring.userNotFound"));

	user.setForgotPasswordCode(UUID.randomUUID().toString());
	userRepository.save(user);

	LemonUtil.afterCommit(() -> {
	    mailForgotPasswordLink(user);
	});
}
```

As you see, it fetches a user by email, sets its `forgotPasswordCode`, and calls `mailForgotPasswordLink` after successful commit. An exception is thrown if there would be no user with the given id.

`mailForgotPasswordLink` composes and sends a mail to the user, containing the link with the *forgotPasswordCode* embedded:

```
protected void mailForgotPasswordLink(U user) {

	try {

		String forgotPasswordLink = properties.getApplicationUrl()
				+ "/users/" + user.getForgotPasswordCode()
```

```
                        + "/reset-password";

            mailSender.send(user.getEmail(),
                  LemonUtil.getMessage(
                        "com.naturalprogrammer.spring.forgotPasswordSubject"),
                  LemonUtil.getMessage(
                        "com.naturalprogrammer.spring.forgotPasswordEmail",
                        forgotPasswordLink));
      } catch (MessagingException e) {
            log.error(ExceptionUtils.getStackTrace(e));
      }
}
```

As you see, if sending the mail fails, we just log the exception and keep silent. That means, if the user does not receive the mail within certain time, he will have to retry the function again.

19 Reset password

See the API documentation to know how it works. The code is discussed below.

Controller method

The controller method is a single liner, receiving *forgotPasswordCode* as a path variable, *newPassword* as a request parameter, and calling the service layer:

```
@PostMapping("/users/{forgotPasswordCode}/reset-password")
public void resetPassword(@PathVariable String forgotPasswordCode,
                          @RequestParam String newPassword) {

      lemonService.resetPassword(forgotPasswordCode, newPassword);
}
```

Service method

It looks like this:

```
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void resetPassword(@Valid @NotBlank String forgotPasswordCode,
                          @Valid @Password String newPassword) {

      U user = userRepository
            .findByForgotPasswordCode(forgotPasswordCode)
            .orElseThrow(() -> MultiErrorException.of(
                  "com.naturalprogrammer.spring.invalidLink"));

      user.setPassword(passwordEncoder.encode(newPassword));
      user.setForgotPasswordCode(null);

      userRepository.save(user);
}
```

Note how

1. Parameters are being validated using annotations
2. Java 8 Optional's *orElseThrow* is being used to throw an exception if the *forgotPasswordCode* is not found.
3. The new password is being encrypted before saving to the database, by the injected *passwordEncoder*.

20 Fetch user by email

Sometimes, your ADMINs or other users might need to view the profile of another user by providing his email id. So, Spring Lemon has a fetch-by-email function, as described in the API documentation. Let's discuss how it's coded.

Controller method

It's a single liner, calling the service method:

```
@GetMapping("/users/fetch-by-email")
public U fetchUserByEmail(@RequestParam String email) {

	return lemonService.fetchUserByEmail(email);
}
```

Note that email is passed as a request paramener.

Service method

The service method looks like this:

```
public U fetchUserByEmail(@Valid @Email @NotBlank String email) {

	// fetch the user
	U user = userRepository.findByEmail(email)
		.orElseThrow(() -> MultiErrorException.of("email",
			"com.naturalprogrammer.spring.userNotFound"));

	// decorate the user, and hide confidential fields
	user.decorate().hideConfidentialFields();

	return user;
}
```

Note that

1. If the optional value returned by `userRepository` will not be having a value, an exception will be thrown. We had discussed about it earlier.
2. Before returning, we decorate the user and then hide its confidential fields.

21 Fetch user by id

This function would fetch a user by his id. See the API documentation for the functionality. Let's discuss its code below.

Controller method

It's a single liner.

```
@GetMapping("/users/{id}")
public U fetchUserById(@PathVariable("id") U user) {

	return lemonService.processUser(user);
}
```

As we know, the path parameter id will be converted to user.

Service method

It looks like this:

```
public U processUser(U user) {

	// ensure that the user exists
	LemonUtil.check("id", user != null,
		"com.naturalprogrammer.spring.userNotFound").go();

	// decorate the user, and hide confidential fields
	user.decorate().hideConfidentialFields();

	return user;
}
```

Looks familiar. Nothing new to learn.

22 Update user

It's the *edit profile* functionality, coded as below.

Controller method

It looks like this:

```
@PutMapping("/users/{id} ")
public U updateUser(@PathVariable("id") U user, @RequestBody U updatedUser) {

	lemonService.updateUser(user, updatedUser);
	return lemonService.userForClient();
}
```

As you see, we are receiving the update data in the second parameter.

Service method

It looks like this:

```
@UserEditPermission
@Validated(AbstractUser.UpdateValidation.class)
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void updateUser(U user, @Valid U updatedUser) {

	LemonUtil.check("id", user != null,
		"com.naturalprogrammer.spring.userNotFound").go();
	LemonUtil.validateVersion(user, updatedUser);

	updateUserFields(user, updatedUser, LemonUtil.getUser());
	userRepository.save(user);
}
```

Note that

1. `@PreAuthorize`, as discussed earlier, will ensure that the current-user has *edit* permission on the given *user* parameter.
2. `@Validated`, as discussed earlier, will ensure that only those fields having *UpdateValidation.class* in their validation group will be validated. So, if you add a new field to a concrete User class, remember to provide this validation group if the field is to be included while editing profile.
3. `validateVersion()`, as discussed earlier, ensures that nobody else has updated it concurrently.
4. For the actual update, we call the `updateUserFields` protected method.

updateUserFields

This method does the actual update.

It looks like this:

```
protected void updateUserFields(U user, U updatedUser, U currentUser) {

	// User is already decorated while checking the 'edit' permission
	// So, user.isRolesEditable() below would work
	if (user.isRolesEditable()) {

		// update the roles

		Set<String> roles = user.getRoles();

		if (updatedUser.isUnverified()) {

			if (!user.hasRole(Role.UNVERIFIED)) {

				makeUnverified(user); // make user unverified
				LemonUtil.afterCommit(() -> sendVerificationMail(user));
			}
		} else {

			if (user.hasRole(Role.UNVERIFIED))
				makeVerified(user); // make user verified
		}

		if (updatedUser.isAdmin())
			roles.add(Role.ADMIN);
		else
			roles.remove(Role.ADMIN);

		if (updatedUser.isBlocked())
			roles.add(Role.BLOCKED);
		else
			roles.remove(Role.BLOCKED);
	}
}
```

Note the following:

1. Only the *roles* field is getting updated here. We don't have any other updatable field in AbstractUser. Well, *email* and *password* are updatable, but we have separate functions for those. If you add more updateable fields to your concrete User class, you may like to override this method. For example, see how a *name* field is added in the getting started guide.
2. If the roles field is indeed editable, we update it as per the flags received from the client through the updatedUser object.

23 Change password

See the API documentation for the usage of the this function. The code is discussed below.

Controller method

It looks like this:

```
@PostMapping("/users/{id}/change-password")
public void changePassword(@PathVariable("id") U user,
      @RequestBody ChangePasswordForm changePasswordForm) {

      lemonService.changePassword(user, changePasswordForm);
}
```

As you see, it receives the data through a *ChangePasswordForm*, and passes that to the service layer.

Service method

The service method looks like this:

```
@UserEditPermission
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void changePassword(U user, @Valid ChangePasswordForm changePasswordForm) {

      // checks
      LemonUtil.check("id", user != null,
            "com.naturalprogrammer.spring.userNotFound").go();
      LemonUtil.check("changePasswordForm.oldPassword",
            passwordEncoder.matches(changePasswordForm.getOldPassword(),
                                                user.getPassword()),
            "com.naturalprogrammer.spring.wrong.password").go();

      // sets the password
      user.setPassword(passwordEncoder.encode(changePasswordForm.getPassword()));
      userRepository.save(user);

      // after successful commit
      LemonUtil.afterCommit(() -> {

            U currentUser = LemonUtil.getUser();

            if (currentUser.equals(user)) { // if current-user's password changed,

                  LemonUtil.logOut(); // log him out
            }
```

```
        });
}
```

Note that

1. ChangePasswordForm, which would be validated, looks like this:

   ```
   @RetypePassword
   public class ChangePasswordForm implements RetypePasswordForm {

         @Password
         private String oldPassword;

         @Password
         private String password;

         @Password
         private String retypePassword;

         ...
   }
   ```

   See that it implements `RetypePasswordForm`, and is annotated with `@RetypePassword`, discussed earlier. This will ensure that both *password* and *retypePassword* would be same.

2. For comparing the *oldPassword* field with the existing database password, `passwordEncoder.matches` is used.
3. The user is logged out on successful commit.

24 Changing email

The email of a user can be changed by following the process described in the API documentation. As you know, there are two requests involved in this process. Both are discussed below.

Requesting changing email

This would be the first request, in which the user will be sent an email containing a link with a secret *change email code*. Its conteoller and service methods are discussed below:

Controller method

It's coded like this:

```
@PostMapping("/users/{id}/request-email-change")
public void requestEmailChange(@PathVariable("id") U user,
                               @RequestBody U updatedUser) {

      lemonService.requestEmailChange(user, updatedUser);
}
```

The `updatedUser` parameter will have the *newEmail* and password fed by the *user*.

Service method

It's coded like this:

```
@UserEditPermission
@Validated(AbstractUser.ChangeEmailValidation.class)
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void requestEmailChange(U user, @Valid U updatedUser) {

      // checks
      LemonUtil.check("id", user != null,
            "com.naturalprogrammer.spring.userNotFound").go();
      LemonUtil.check("updatedUser.password",
            passwordEncoder.matches(
                  updatedUser.getPassword(),
                  LemonUtil.getUser().getPassword()),
            "com.naturalprogrammer.spring.wrong.password").go();

      // preserves the new email id
      user.setNewEmail(updatedUser.getNewEmail());
      user.setChangeEmailCode(UUID.randomUUID().toString());
      userRepository.save(user);

      // after successful commit, mails a link to the user
```

```
        LemonUtil.afterCommit(() -> mailChangeEmailLink(user));
}
```

Note that for receiving the *newEmail* and *password* fed by the user, we are using the entire User entity instead of a DTO. It's okay, because the `@Validated` annotation will restrict validations only to the newEmail and password fields, because of the use of the `ChangeEmailValidation.class` validation group.

Because the *newEmail* field in *AbstractUser* is annotated with `@UniqueEmail`, it will be ensured that there is no existing user having the same email id.

After successful commit, a mail to the user is sent by using the `mailChangeEmailLink` method, which is coded as below:

```
protected void mailChangeEmailLink(U user) {

        try {

                String changeEmailLink = properties.getApplicationUrl()
                    + "/users/" + user.getChangeEmailCode()
                    + "/change-email";

                mailSender.send(user.getEmail(),
                        LemonUtil.getMessage(
                                "com.naturalprogrammer.spring.changeEmailSubject"),
                        LemonUtil.getMessage(
                                "com.naturalprogrammer.spring.changeEmailEmail",
                                 changeEmailLink));

        } catch (MessagingException e) {
                log.error(ExceptionUtils.getStackTrace(e));
        }
}
```

Changing email

This function will be used when the user clicks on the link mailed to him after the above request. Below are its controller and service methods.

Controller method

It looks like this:

```
@PostMapping("/users/{changeEmailCode}/change-email")
public void changeEmail(@PathVariable String changeEmailCode) {

        lemonService.changeEmail(changeEmailCode);
}
```

Service method

It looks like this:

```
@PreAuthorize("isAuthenticated()")
@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void changeEmail(@Valid @NotBlank String changeEmailCode) {

	// fetch the current-user
	U currentUser = LemonUtil.getUser();
	U user = userRepository.findOne(currentUser.getId());

	// checks

	LemonUtil.check(changeEmailCode.equals(user.getChangeEmailCode()),
			"com.naturalprogrammer.spring.wrong.changeEmailCode").go();

	// Ensure that the email would be unique
	LemonUtil.check(
			!userRepository.findByEmail(user.getNewEmail()).isPresent(),
			"com.naturalprogrammer.spring.duplicate.email").go();

	// update the fields
	user.setEmail(user.getNewEmail());
	user.setNewEmail(null);
	user.setChangeEmailCode(null);

	// make the user verified if he is not
	if (user.hasRole(Role.UNVERIFIED))
		makeVerified(user);

	userRepository.save(user);

	// logout after successful commit
	LemonUtil.afterCommit(LemonUtil::logOut);
}
```

Note that we are re-ensuring the uniqueness of *newEmail*, to check if somebody has joined meanwhile using the same email id.

25 Creating an Admin

When an application is installed, it's helpful to have its database initialized with an ADMIN user. In Spring Lemon, we do it inside an `ApplicationReadyEvent` handler, which is coded in the *LemonService* class, as below:

```
@EventListener
public void afterApplicationReady(ApplicationReadyEvent event) {

	onStartup(); // delegate to onStartup()
}


@Transactional(propagation=Propagation.REQUIRED, readOnly=false)
public void onStartup() {

	try {
		// Check if the user already exists
		userDetailsService
			.loadUserByUsername(properties.getAdmin().getUsername());

	} catch (UsernameNotFoundException e) {

		// Doesn't exist. So, create it.
		U user = createAdminUser();
		userRepository.save(user);
	}
}


protected U createAdminUser() {

	Admin initialAdmin = properties.getAdmin();

	U user = newUser();
	user.setUsername(initialAdmin.getUsername());
	user.setPassword(passwordEncoder.encode(
		properties.getAdmin().getPassword()));
	user.getRoles().add(Role.ADMIN);

	return user;
}
```

The `@EventListener` annotation tells Spring to call the annotated method when the type of the event that is passed as parameter occurs. In this case, the type is `ApplicationReadyEvent` – i.e. after the application starts and becomes ready to serve requests. More about Spring Boot events can be found here.

As you see in the code above, we refer to application properties for getting the username and password of the user to be created. So, *application*.properties* can have lines as below:

```
lemon.admin.username: admin@example.com
lemon.admin.password: admin-password
```

In the *onStartup* method above, we first check whether a user with the given username already exists. It would not exist when the application is run first time after installation, and so the user would get created.

26 Testing

If you look at Spring and Spring Boot reference material, you will find that Spring provides multiple ways to write unit and integration tests. Among those, writing JUnit integration tests seems like a good choice for REST APIs in general. Using RestAssured makes it more elegant. We have done the same for the Lemon Demo Application, and in this chapter, we will briefly discuss it. Covering testing elaborately is out of scope of this book, but we will provide some basic guidance for you to get up and running.

> **What is integration testing**
>
> In layman terms, whereas unit testing would mean testing individual methods or components of your API, integration testing would mean testing the API by actually running it and making calls to it.
>
> **What is RestAssured**
>
> When doing integration testing, we can use Spring's TestRestTemplate to make calls to the API. But we find RestAssured as a better alternative – it makes writing test cases easier and more readable.

Dependencies needed

The following dependencies would be needed in *pom.xml* for doing automated testing and using *RestAssured*:

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

<dependency>
    <groupId>com.jayway.restassured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>RELEASE</version>
    <scope>test</scope>
</dependency>
...
```

The first one normally comes by default when we create a new project by using STS or start.spring.io, whereas we the second one will have to be added manually.

Coding a test class

Our JUnit integration test classes would look like this:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=RANDOM_PORT)
@ActiveProfiles("itest")
public class SampleTests {

    @BeforeClass
    public static void init() {
      // this code will run once, before any of the test
      // of this class is run
    }

    @Before
    public final void setUp() {
      // this code will everytime before a test case
      // of this class is run
    }

    @After
    public final void tearDown() {
      // this code will run everytime after a test case
      // of this class is run
    }

    @AfterClass
    public static void destroy() {
      // this code will run once, after all the tests
      // of this class are run
    }

    @Test
    public void testCase1() {
      // a test case
    }

    @Test
    public void testCase2() {
      // another test case
    }
}
```

The comments inside the methods are self-explanatory, but the *@ActiveProfiles* annotation needs some explanation, which we discuss next.

Using @ActiveProfiles for in-memory database etc.

Integration testing needs the application to be run, so that test cases can make HTTP requests to it. You may like to customize how the application is run while testing. For example, you may like to use an in-memory database instead of MySQL.

Having a distinct profile, say *itest*, looks like a good way to achieve this. The `@ActiveProfiles` annotation used above tells Spring to set the active profiles to *itest* while testing. So, for using an in memory HSQL database instead of MySQL, we can have an *application-itest.properties* containing the following line:

```
spring.jpa.database: HSQL
```

As discussed earlier, to automatically choose MySQL or HSQL depending on environment, we should have added the MySQL properties in other properties files, such as *application-dev.properties*, *application-prod.properties*, etc., as below:

```
spring.jpa.database: MYSQL
spring.jpa.hibernate.ddl-auto: update

spring.datasource.url: jdbc:mysql://localhost:3306/spring-angular
spring.datasource.username: spring
spring.datasource.password: spring
```

Also, our *application.properties* could set the default active profiles to *dev*:

```
spring.profiles.active: dev
```

Of course, HSQL should be available in the class path, for this to work. That means, the dependencies section of our *pom.xml* should contain HSQL with *test* scope:

```
...
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>test</scope>
</dependency>
...
```

Configuring RestAssured

Setting HTTP port

RestAssured sends HTTP requests to the application, and for that it needs to know the port on which the application is running. Setting its static `port` variable, as below, is needed for that:

```
RestAssured.port = 8080;
```

The above code assumes that the application would be running on 8080. But, it's a good practice for test applications to use a randomly free port instead of 8080. The `@SpringBootTest(webEnvironment=RANDOM_PORT)` annotation on a test class ensures that. Inside a test class, the actual port number can be injected, and RestAssured.port can be set, as below:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=RANDOM_PORT)
@ActiveProfiles("itest")
public class SampleTests {

    ...

    @LocalServerPort
    public void setPort(int port) {
       RestAssured.port = port;
    }

    ...
}
```

Refer here to know more about how to discover the HTTP port at runtime,.

Logging

Sometimes we may like to see the contents of the request and the response while running test cases. The RestAssured reference material has described how to do it in details. In particular, we would normally want to enable logging of the request and response for all requests, if validation fails. This can be configured as below:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SpringSampleAppApplication.class)
@WebIntegrationTest
@ActiveProfiles("itest")
public class SampleTests {

    ...

    @BeforeClass
    public static void init() {

       RestAssured.enableLoggingOfRequestAndResponseIfValidationFails();
    }

    ...
}
```

Using RestAssured Filters

RestAssured lets you make HTTP requests in an elegant manner. Here is a simple example from its documentation:

```
get("/lotto").then().body("lotto.lottoId", equalTo(5));
```

The above is a simple GET request. But, when a request needs authentication or CSRF protection, you may need to mimic a browser – i.e. send JSESSIONID and other needed cookies/headers obtained in previous requests.

Rather than sending the cookies and headers manually in each request, we can use RestAssured filters for this. In fact, RestAssured already provides a SessionFilter which remembers the session, thus supporting session based authentication and other features. We coded a couple of other filters in our lemon demo application - a JsonPrefixFilter for stripping out the JSON Vulnerability characters, and a XsrfFilter for CSRF.

Instead of specifying these filters in each request, we can create a request specification by using a request specification builder. In the lemon demo application, the `configureFilters` utility method in the MyTestUtil class creates our specification, as below:

```
public static RequestSpecification configureFilters() {

	return new RequestSpecBuilder()
		.addFilter(new SessionFilter())
		.addFilter(new XsrfFilter())
		.addFilter(new JsonPrefixFilter())
		.build();
}
```

Inside a test case, this can then be used to create a spec and use it, as below:

```
RequestSpecification filters = MyTestUtil.configureFilters();

given()
    .spec(filters)
.get("/api/core/ping")
...
```

Truncating/initializing HSQL

Our test cases should be independent of one another. We should thus roll the database back to original position after a test case is run. But, rolling back transactions isn't possible when doing integration tests. However, we can very well truncate the database using SQL. For example, in the lemon demo application, we have a `truncateDb` utility method in MyTestUtils class which truncates an HSQL database:

```
public static void truncateDb(DataSource dataSource) throws SQLException {

    try (
            Connection connection = dataSource.getConnection();
            Statement databaseTruncationStatement = connection.createStatement();
    ) {
            Assert.assertTrue(
```

```
            "This @After method wipes the entire database!",
            connection.getMetaData().getDriverName()
                  .equals("HSQL Database Engine Driver"));

        databaseTruncationStatement.executeUpdate(
            "TRUNCATE SCHEMA public AND COMMIT");
    }
}
```

The above method would assert that the given data source is indeed an HSQL database, and then truncate it. Note that the connection and statement are declared as resources, which would get auto-closed after the try block finishes.

So, inject the data source and call the above routine in an `@After` method for doing the truncation:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SpringSampleAppApplication.class)
@WebIntegrationTest
@ActiveProfiles("itest")
public class SampleTests {

    ...

    @Autowired
    protected DataSource dataSource;

    @After
    public final void baseTearDown() throws SQLException {

        MyTestUtil.truncateDb(dataSource);
    }

    ...
}
```

Similarly, any database initialization code can be put in a `@Before` method. For example, the lemon demo application has a @Before method as below:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = SpringSampleAppApplication.class)
@WebIntegrationTest
@ActiveProfiles("itest")
public class SampleTests {

    ...

    @Autowired
    protected MyService service;

    @Before
    public final void baseSetUp() {

       service.onStartup();
```

```
    }

    ...
}
```

If you recall, the onStartup method creates the first Admin user in a new database.

Preventing parallel execution of test cases

For the database truncation strategy to work properly, our test cases should execute one after another, instead of running in parallel. It's anyway the default behavior of JUnit, so we shouldn't need to do anything by default. But, if we want to doubly ensure it to avoid some case where the defaults might have been overridden by something else, we can use a global Lock.

The next section shows how such a lock is used in the lemon demo application.

Creating a base test class

Instead of coding in each test class all the above that we discussed, having a base test class would be handy. In the lemon demo application, we have an `AbstractTests` class as below, and all our test classes extend from it.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment=RANDOM_PORT)
@ActiveProfiles("itest")
public abstract class AbstractTests {

    @Autowired
    protected LemonProperties lemonProperties;

    @Autowired
    protected DataSource dataSource;

    @Autowired
    protected MyService service;

    @LocalServerPort
    public void setPort(int port) {
       RestAssured.port = port;
    }

    @BeforeClass
    public static void init() {
       RestAssured.enableLoggingOfRequestAndResponseIfValidationFails();
    }

    private static Lock sequential = new ReentrantLock();

    protected RequestSpecification filters;
```

```
    @Before
    public final void baseSetUp() {

       sequential.lock();
       service.onStartup();
       filters = MyTestUtil.configureFilters();
    }

    @After
    public final void baseTearDown() throws SQLException {

        filters = null;
        MyTestUtil.truncateDb(dataSource);
        sequential.unlock();
    }
}
```

Note how we have used a `Lock` to prevent parallel execution of test cases. Note also that we have the `filters` as an instance variable, and we initialize it in a @Before method – i.e. a fresh instance of `filters` is used in every test case.

Coding test cases

Test cases can be written in `@Test` methods inside classes inherited from our `AbstractTests` class that we discussed above. How to write the test cases is well explained in the RestAssured documentation – so we would not cover that here. To learn more about it, refer to the test cases coded in the lemon demo application. Notice there that we have coded a `hasErrors` custom matcher in MyTestUtil class, for checking the response containing field errors thrown from the application. How to write custom Hamcrest matchers is out of scope of this book, but you can copy our `hasErrors` matcher and use it in your test cases, even without understanding in details how it's coded.

27 Stay in touch

So, we have come to the end of the course. Do follow our blog, www.naturalprogrammer.com, for new posts and updates. Please let us know your suggestions and feedback, and bugs or error you discover in our books and related projects. The Help and Support section has details on how to stay in touch.