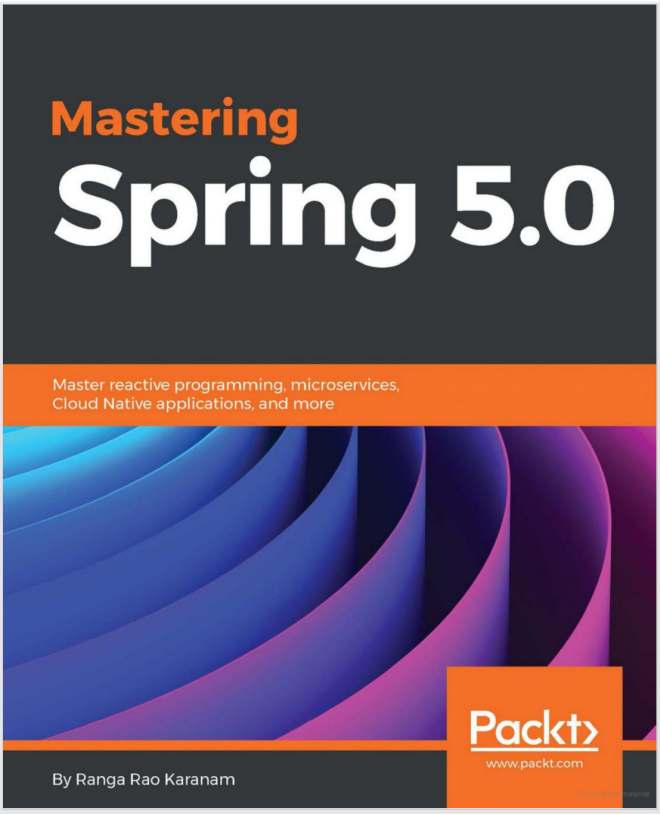



# Mastering Spring 5.0

Master reactive programming, microservices, Cloud Native applications, and more

*This book is based on Spring Version 5.0 RC1*

**Ranga Rao Karanam**

Packt

BIRMINGHAM - MUMBAI



## Mastering Spring 5.0

First published: June 2017

Production reference: 1240617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78712-317-5

www.packtpub.com





# www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www.packtpub.com/mapt

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser





# Table of Contents

# Preface

Spring 5.0 is due to arrive with a myriad of new and exciting features that will change the way we've used the framework so far. This book will show you this evolution--from solving the problems of testable applications to building distributed applications on the Cloud.
The book begins with an insight into the new features in Spring 5.0, and shows you how to build an application using Spring MVC. You will then get a thorough understanding of how to build and extend microservices using the Spring Framework. You will also understand how to build and deploy Cloud applications. You will realize how application architectures have evolved from monoliths to those built around microservices. The advanced features of Spring Boot will also be covered and displayed through powerful examples.
By the end of this book, you will be equipped with the knowledge and best practices to develop applications with the Spring Framework.

## What this book covers

`Chapter 1`, *Evolution to Spring Framework 5.0*, takes you through the evolution of the Spring Framework, ranging from its initial versions to Spring 5.0. Initially, Spring was used to develop testable applications using dependency injection and core modules. Recent Spring Projects, such as Spring Boot, Spring Cloud, Spring Cloud Data Flow--deal with application infrastructure and moving applications to Cloud. We get an overview of different Spring modules and projects.

`Chapter 2`, *Dependency Injection*, dives deep into dependency injection. We will look at different kinds of dependency injection methods available in Spring, and how auto-wiring makes your life easy. We will also take a quick look into unit testing.

`Chapter 3`, *Building a Web Application with Spring MVC*, gives a quick overview of building a web application with Spring MVC.

`Chapter 4`, *Evolution toward Microservices and Cloud-Native Applications*, explains the evolution of application architectures in the last decade. We will understand why microservices and Cloud Native applications are needed and get a quick overview of the different Spring projects that help us build Cloud-Native applications.





`Chapter 5`, *Building Microservices with Spring Boot*, discusses how Spring Boot takes away the complexity in creating production-grade Spring-based applications. It makes it easy to get started with Spring-based projects and provides easy integration with third-party libraries. In this chapter, we will take the students on a journey with Spring Boot. We will start with implementing a basic web service and then move on to adding caching, exception handling, HATEOAS, and internationalization, while making use of different features from the Spring Framework.

`Chapter 6`, *Extending Microservices*, focuses on adding more advanced features to the microservices that we built in `Chapter 4`, *Evolution toward Microservices and Cloud-Native Applications*.

`Chapter 7`, *Advanced Spring Boot Features*, takes a look at the advanced features in Spring Boot. You will learn how to monitor a microservice with a Spring Boot Actuator. Then, you will deploy the microservice to Cloud. You will also learn how to develop more effectively with the developer tools provided by Spring Boot.

`Chapter 8`, *Spring Data*, discusses the Spring Data module. We will develop simple applications to integrate Spring with JPA and Big Data technologies.

`Chapter 9`, *Spring Cloud*, discusses the distributed systems in the Cloud that have common problems, configuration management, service discovery, circuit breakers, and intelligent routing. In this chapter, you will learn how Spring Cloud helps you develop solutions for these common patterns. These solutions should work well on the Cloud as well as

these collision patterns. These solutions should work well on the Cloud as well as developer local systems.

`Chapter 10`, *Spring Cloud Data Flow*, talks about the Spring Cloud Data Flow, which offers a collection of patterns and best practices for microservices-based distributed streaming and batch data pipelines. In this chapter, we will understand the basics of Spring Cloud Data Flow and use it to build basic data flow use cases.

`Chapter 11`, *Reactive Programming*, explores programming with asynchronous data streams. In this chapter, we will understand Reactive programming and take a quick look at the features provided by the Spring Framework.

`Chapter 12`, *Spring Best Practices*, helps you understand best practices in developing enterprise applications with Spring related to unit testing, integration testing, maintaining Spring configuration, and more.

`Chapter 13`, *Working with Kotlin in Spring*, introduces you to a JVM language gaining quick popularity--Kotlin. We will discuss how to setup a Kotlin project in Eclipse. We will create a new Spring Boot project using Kotlin and implement a couple of basic services with unit and integration testing.







## What you need for this book

To be able to run examples from this book, you will need the following tools:

- Java 8
- Eclipse IDE
- Postman

We will use Maven embedded into Eclipse IDE to download all the dependencies that are needed.

## Who this book is for

This book is for experienced Java developers who knows the basics of Spring, and wants to learn how to use Spring Boot to build applications and deploy them to the Cloud.

## Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows: "Configure `spring-boot-starter-parent` in your `pom.xml` file".

A block of code is set as follows:

```
<properties>
  <mockito.version>1.10.20</mockito.version>
</properties>
```

Any command-line input or output is written as follows:

```
mvn clean install
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Provide the details and click on **Generate Project**."







Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support`and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.

6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Spring-5.0`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support`and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.







Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

## Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.





Pages 7 to 40 are not shown in this preview.



We will want to create a mock for `DataService`. There are multiple approaches to creating mocks with Mockito. Let's use the simplest among them--annotations. We use the `@Mock` annotation to create a mock for `DataService`:

```
@Mock
private DataService dataService;
```

Once we create the mock, we will need to inject it into the class under test, `BusinessServiceImpl`. We do that using the `@InjectMocks` annotation:

```
@InjectMocks
private BusinessService service =
new BusinessServiceImpl();
```

In the test method, we will need to stub the mock service to provide the data that we want it to provide. There are multiple approaches. We will use the BDD style methods provided by

to provide. There are multiple approaches. We will use the BDD style methods provided by Mockito to mock the `retrieveData` method:

```
BDDMockito.given(dataService.retrieveData(
  Matchers.any(User.class)))
  .willReturn(Arrays.asList(new Data(10),
  new Data(15), new Data(25)));
```

What we are defining in the preceding code is called stubbing. As with anything with Mockito, this is extremely readable. When the `retrieveData` method is called on the `dataService` mock with any object of type `User`, it returns a list of three items with values specified.

When we use Mockito annotations, we would need to use a specific JUnit runner, that is, `MockitoJunitRunner`. `MockitoJunitRunner` helps in keeping the test code clean and provides clear debugging information in case of test failures. `MockitoJunitRunner` initializes the beans annotated with `@Mock` annotation and also validates the usage of framework after execution of each test method.

```
@RunWith(MockitoJUnitRunner.class)
```

The complete list of the test is as follows. It has one test method:

```
@RunWith(MockitoJUnitRunner.class)
public class BusinessServiceMockitoTest {
  private static final User DUMMY_USER = new User("dummy");
   @Mock
  private DataService dataService;
  @InjectMocks
  private BusinessService service =
  new BusinessServiceImpl();
```







```
  @Test
  public void testCalculateSum() {
    BDDMockito.given(dataService.retrieveData(
    Matchers.any(User.class)))
    .willReturn(
       Arrays.asList(new Data(10),
       new Data(15), new Data(25)));
       long sum = service.calculateSum(DUMMY_USER);
       assertEquals(10 + 15 + 25, sum);
    }
  }
```

## Container managed beans

Instead of a class creating its own dependencies, in the earlier example, we looked at how the Spring IoC container can take over the responsibility of managing beans and their dependencies. The beans that are managed by the container are called **Container Managed Beans**.








Delegating the creation and management of beans to the container has many advantages. Some of them are listed as follows:

- Since classes are not responsible for creating dependencies, they are loosely coupled and testable. This leads to good design and fewer defects.
- Since the container manages the beans, a few hooks around the beans can be introduced in a more generic way. Cross-cutting concerns, such as logging, caching, transaction management, and exception handling can be woven around these beans using **Aspect-Oriented Programming** (**AOP**). This leads to more maintainable code.

## Dependency injection types

In the previous example, we used a setter method to wire in the dependency. There are two types of dependency injections that are used frequently:

- The setter injection
- The constructor injection

### The setter injection

The setter injection is used to inject the dependencies through setter methods. In the following example, the instance of `DataService` uses the setter injection:

```
public class BusinessServiceImpl {
  private DataService dataService;
  @Autowired
  public void setDataService(DataService dataService) {
    this.dataService = dataService;
  }
}
```

Actually, in order to use the setter injection, you do not even need to declare a setter method. If you specify `@Autowired` on the variable, Spring automatically uses the setter injection. So, the following code is all that you need for the setter injection for `DataService`:

```
public class BusinessServiceImpl {
  @Autowired
```

```
  private DataService dataService;
}
```







## The constructor injection

The constructor injection, on the other hand, uses a constructor to inject dependencies. The following code shows how to use a constructor for injecting in `DataService`:

```
public class BusinessServiceImpl {
  private DataService dataService;
  @Autowired
  public BusinessServiceImpl(DataService dataService) {
    super();
    this.dataService = dataService;
  }
}
```

When you run the code with the preceding implementation of `BusinessServiceImpl`, you will see this statement in the log, asserting that autowiring took place using the constructor:

```
Autowiring by type from bean name 'businessServiceImpl' via
constructor to bean named 'dataServiceImpl'
```

## Constructor versus setter injection

Originally, in XML-based application contexts, we used the constructor injection with mandatory dependencies and the setter injection with nonmandatory dependencies.

However, an important thing to note is that when we use `@Autowired` on a field or a method, the dependency is required by default. If no candidates are available for an `@Autowired` field, autowiring fails and throws an exception. So, the choice is not so clear anymore with Java application contexts.

Using the setter injection results in the state of the object changing during the creation. For fans of immutable objects, the constructor injection might be the way to go. Using the setter injection might sometimes hide the fact that a class has a lot of dependencies. Using the constructor injection makes it obvious, since the size of the constructor increases.

# Spring bean scopes

Spring beans can be created with multiple scopes. The default scope is a singleton.

Since there is only one instance of a singleton bean, it cannot contain any data that is specific to a request.





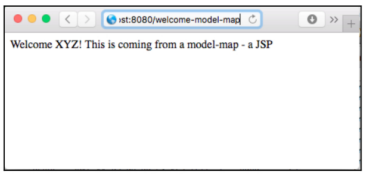




One thing to note is this:

- `${name}`: This uses the **Expression Language** (**EL**) syntax to access the attribute from the model.

Here is a screenshot of how this would look on the screen when the URL is hit:




## Unit testing

In this unit test, we want to set up `BasicModelMapController`, execute a get request to `/welcome-model-map`, and check whether the model has the expected attribute and whether the expected view name is returned.

### Setting up the Controller to test

This step is very similar to the previous flow. We instantiate Mock MVC with `BasicModelMapController`:

```
this.mockMvc = MockMvcBuilders.standaloneSetup
  (new BasicModelMapController())
  .setViewResolvers(viewResolver()).build();
```

### Writing the Test method

The complete Test method is shown in the following code:

```
@Test
public void basicTest() throws Exception {
  this.mockMvc
  .perform(
```







```
    get("/welcome-model-map")
    .accept(MediaType.parseMediaType
    ("application/html;charset=UTF-8")))
    .andExpect(model().attribute("name", "XYZ"))
    .andExpect(view().name("welcome-model-map"));
}
```

A few important things to note:

- `get("/welcome-model-map")`: Execute `get` request to the specified URL
- `model().attribute("name", "XYZ")`: Result Matcher to check if the model contains specified attribute **name** with specified value **XYZ**
- `view().name("welcome-model-map")`: Result Matcher to check if the view name returned is as specified

## Flow 4 - Controller redirecting to a View with ModelAndView

In the previous flow, we returned a view name and populated the model with attributes to be used in the view. Spring MVC provides an alternate approach using `ModelAndView`. The controller method can return a `ModelAndView` object with the view name and appropriate attributes in the Model. In this flow, we will explore this alternate approach.

### Spring MVC controller

Take a look at the following controller:

```
@Controller
public class BasicModelViewController {
 @RequestMapping(value = "/welcome-model-view")
  public ModelAndView welcome(ModelMap model) {
    model.put("name", "XYZ");
    return new ModelAndView("welcome-model-view", model);
  }
}
```







A few important things to note are as follows:

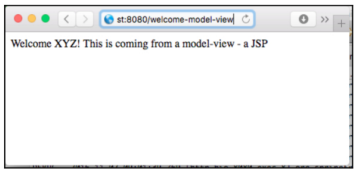
- `@RequestMapping(value = "/welcome-model-view")`: The URI mapped is `/welcome-model-view`.
- `public ModelAndView welcome(ModelMap model)`: Note that the return value is no longer a String. It is `ModelAndView`.
- `return new ModelAndView("welcome-model-view", model)`: Create a `ModelAndView` object with the appropriate view name and model.

### Creating a View

Let's create a view using the model attribute `name` that was set in the model in the controller. Create a simple JSP in the `/WEB-INF/views/welcome-model-view.jsp` path:

```
Welcome ${name}! This is coming from a model-view - a JSP
```

Here is a screenshot of how this would look on the screen when the URL is hit:




### Unit testing

Unit testing for this flow is similar to the previous flow. We would need to check if the expected view name is returned.







## Flow 5 - Controller redirecting to a View with a form

Now let's shift our attention to creating a simple form to capture input from the user.

The following steps will be needed:

- Create a simple POJO. We want to create a user. We will create a POJO User.
- Create a couple of Controller methods--one to display the form, and the other to capture the details entered in the form.
- Create a simple View with the form.

### Creating a command or form backing object

POJO stands for Plain Old Java Object. It is usually used to represent a bean following the typical JavaBean conventions. Typically, it contains private member variables with getters and setters and a no-argument constructor.

We will create a simple POJO to act as a command object. Important parts of the class are listed as follows:

```
public class User {
  private String guid;
  private String name;
  private String userId;
  private String password;
  private String password2;
  //Constructor
  //Getters and Setters
  //toString
}
```

A few important things to note are as follows:

- This class does not have any annotations or Spring-related mappings. Any bean can act as a form-backing object.
- We are going to capture the `name`, `user ID`, and `password` in the form. We have a password confirmation field, `password2`, and unique identifier field guid.
- `Constructor`, `getters`, `setters`, and `toString` methods are not shown for brevity.









- **Implicit determination of the View**: If a view name is not present in the return type, it is determined using `DefaultRequestToViewNameTranslator`. By default, `DefaultRequestToViewNameTranslator` removes the leading and trailing slashes as well as the file extension from the URI; for example, the `display.html` becomes display.

The following are some of the return types that are supported on Controller methods with Request Mapping:

| Return Type | What happens? |
|---|---|
| ModelAndView | The object includes a reference to the model and the view name. |
| Model | Only Model is returned. The view name is determined using `DefaultRequestToViewNameTranslator`. |
| Map | A simple map to expose a model. |
| View | A view with a model implicitly defined. |
| String | Reference to a view name. |

## View resolution

Spring MVC provides very flexible view resolution. It provides multiple view options:

- Integration with JSP, Freemarker.
- Multiple view resolution strategies. A few of them are listed as follows:
  - `XmlViewResolver`: View resolution based on an external XML configuration
  - `ResourceBundleViewResolver`: View resolution based on a property file
  - `UrlBasedViewResolver`: Direct mapping of the logical view name to a URL
  - `ContentNegotiatingViewResolver`: Delegates to other view resolvers based on the Accept request header
- Support for chaining of view resolvers with the explicitly defined order of preference.
- Direct generation of XML, JSON, and Atom using Content Negotiation.







## Configuring JSP view resolver

The following example shows the commonly used approach to configure a JSP view resolver using `InternalResourceViewResolver`. The physical view name is determined using the configured prefix and suffix for the logical view name using `JstlView`:

```
<bean id="jspViewResolver" class=
  "org.springframework.web.servlet.view.
  InternalResourceViewResolver">
  <property name="viewClass"
    value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

There are other approaches using property and XML files for mapping.

## Configuring Freemarker

The following example shows the typical approach used to configure a Freemarker view resolver.

First, the `freemarkerConfig` bean is used to load the Freemarker templates:

```
<bean id="freemarkerConfig"
  class="org.springframework.web.servlet.view.
  freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-
  INF/freemarker/"/>
</bean>
```

The following bean definition shows how to configure a Freemarker view resolver:

```
<bean id="freemarkerViewResolver"
 class="org.springframework.web.servlet.view.
 freemarker.FreeMarkerViewResolver">
   <property name="cache" value="true"/>
   <property name="prefix" value=""/>
   <property name="suffix" value=".ftl"/>
</bean>
```

As with JSPs, the view resolution can be defined using properties or an XML file.

## Handler mappings and Interceptors

In the version before Spring 2.5 (before there was support for Annotations), the mapping between a URL and a Controller (also called a handler) was expressed using something called a handler mapping. It is almost a historical fact today. The use of annotations eliminated the need for an explicit handler mapping.

HandlerInterceptors can be used to intercept requests to handlers (or **controllers**). Sometimes, you would want to do some processing before and after a request. You might want to log the content of the request and response, or you might want to find out how much time a specific request took.

There are two steps in creating a HandlerInterceptor:

1. Define the HandlerInterceptor.
2. Map the HandlerInterceptor to the specific handlers to be intercepted.

### Defining a HandlerInterceptor

The following are the methods you can override in `HandlerInterceptorAdapter`:

- `public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)`: Invoked before the handler method is invoked
- `public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)`: Invoked after the handler method is invoked
- `public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)`: Invoked after the request processing is complete

The following example implementation shows how to create a HandlerInterceptor. Let's start with creating a new class that extends `HandlerInterceptorAdapter`:

```
public class HandlerTimeLoggingInterceptor extends
HandlerInterceptorAdapter {
```





Pages 86 to 96 are not shown in this preview.



A few things to note are as follows:

- `@RunWith(SpringRunner.class)`: `SpringRunner` helps us launch a Spring context.
- `@WebAppConfiguration`: Used to launch a web app context with Spring MVC
- `@ContextConfiguration("file:src/main/webapp/WEB-INF/user-web-context.xml")`: Specifies the location of the spring context XML.
- `this.mockMvc = MockMvcBuilders.webAppContextSetup(this.wac).build()`: In the earlier examples, we used a standalone setup. However, in this example, we want to launch the entire web app. So, we use `webAppContextSetup`.
- The execution of the test is very similar to how we did it in earlier tests.

## Serving static resources

Most teams today have separate teams delivering frontend and backend content. The frontend is developed with modern JavaScript frameworks, such as AngularJs, Backbone, and so on. Backend is built through web applications or REST services based on frameworks such as Spring MVC.

With this evolution in frontend frameworks, it is very important to find the right solutions to version and deliver frontend static content.

The following are some of the important features provided by the Spring MVC framework:

- They expose static content from folders in the web application root
- They enable caching
- They enable GZip compression of static content

### Exposing static content

Web applications typically have a lot of static content. Spring MVC provides options to expose static content from folders on the web application root as well locations on the classpath. The following snippet shows that content within the war can be exposed as static content:

```
<mvc:resources
mapping="/resources/**"
location="/static-resources/"/>
```







Things to note are as follows:

- `location="/static-resources/"`: The location specifies the folders inside the war or classpath that you would want to expose as static content. In this example, we want to expose all the content in the `static-resources` folder inside the root of war as static content. We can specify multiple comma-separated values to expose multiple folders under the same external facing URI.
- `mapping="/resources/**"`: The mapping specifies the external facing URI path. So, a CSS file named `app.css` inside the static-resources folder can be accessed using the `/resources/app.css` URI.

The complete Java configuration for the same configuration is shown here:

```
@Configuration
@EnableWebMvc
public class WebConfig extends WebMvcConfigurerAdapter {
  @Override
  public void addResourceHandlers
```

```
    (ResourceHandlerRegistry registry) {
      registry
     .addResourceHandler("/static-resources/**")
     .addResourceLocations("/static-resources/");
    }
}
```

## Caching static content

Caching for static resources can be enabled for improved performance. The browser would cache the resources served for the specified time period. The `cache-period` attribute or the `setCachePeriod` method can be used to specify the caching interval (in seconds) based on the type of configuration used. The following snippets show the details:

This is the Java configuration:

```
registry
.addResourceHandler("/resources/**")
.addResourceLocations("/static-resources/")
.setCachePeriod(365 * 24 * 60 * 60);
```

This is the XML configuration:

```
<mvc:resources
 mapping="/resources/**"
 location="/static-resources/"
 cache-period="365 * 24 * 60 * 60"/>
```







The `Cache-Control: max-age={specified-max-age}` response header will be sent to the browser.

## Enabling GZip compression of static content

Compressing a response is a simple way to make web applications faster. All modern browsers support GZip compression. Instead of sending the full static content file, a compressed file can be sent as a response. The browser will decompress and use the static content.

The browser can specify that it can accept the compressed content with a request header. If the server supports it, it can deliver the compressed content--again, marked with a response header.

Request Header sent from browser is as follows:

```
Accept-Encoding: gzip, deflate
```

Response Header sent from the web application is as follows:

```
Content-Encoding: gzip
```

The following snippet shows how to add a Gzip resolver to deliver compressed static content:

```
registry
  .addResourceHandler("/resources/**")
  .addResourceLocations("/static-resources/")
  .setCachePeriod(365 * 24 * 60 * 60)
  .resourceChain(true)
  .addResolver(new GzipResourceResolver())
  .addResolver(new PathResourceResolver());
```

Things to note are as follows:

- `resourceChain(true)`: We would want to enable Gzip compression, but would want to fall back to delivering the full file if full file was requested. So, we use resource chaining (chaining of resource resolvers).
- `addResolver(new PathResourceResolver()): PathResourceResolver`: This is the default resolver. It resolves based on the resource handlers and locations configured.
- `addResolver(new GzipResourceResolver()): GzipResourceResolver`: This enables Gzip compression when requested.









```
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Now, all requests to our web application will go through the filter. However, we have not configured anything related to security yet. Let's use a simple Java configuration example:

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends
WebSecurityConfigurerAdapter {
  @Autowired
  public void configureGlobalSecurity
  (AuthenticationManagerBuilder auth) throws Exception {
  auth
  .inMemoryAuthentication()
  .withUser("firstuser").password("password1")
  .roles("USER", "ADMIN");
 }
 @Override
 protected void configure(HttpSecurity http)
 throws Exception {
   http
  .authorizeRequests()
  .antMatchers("/login").permitAll()
  .antMatchers("/*secure*/**")
  .access("hasRole('USER')")
  .and().formLogin();
 }
}
```

Things to note are as follows:

- `@EnableWebSecurity`: This annotation enables any Configuration class to contain the definition of Spring Configuration. In this specific instance, we override a couple of methods to provide our specific Spring MVC configuration.
- `WebSecurityConfigurerAdapter`: This class provides a base class to create a Spring configuration (`WebSecurityConfigurer`).
- `protected void configure(HttpSecurity http)`: This method provides the security needs for different URLs.
- `antMatchers("/*secure*/**").access("hasRole('USER')")`: You would need a role of USER to access any URL containing the sub-string `secure`.

- `antMatchers("/login").permitAll()`: Permits access to the login page to all users.







- `public void configureGlobalSecurity(AuthenticationManagerBuilder auth)`: In this example, we are using in-memory authentication. This can be used to connect to a database (`auth.jdbcAuthentication()`), or an LDAP(`auth.ldapAuthentication()`), or a custom authentication provider (created extending `AuthenticationProvider`).
- `withUser("firstuser").password("password1")`: Configures an in-memory valid user ID and password combination.
- `.roles("USER", "ADMIN")`: Assigns roles to the user.

When we try to access any secure URLs, we will be redirected to a login page. Spring Security provides ways of customizing the Logic page as well as the redirection. Only authenticated users with the right roles will be allowed to access the secured application pages.

## Logout

Spring Security provides features to enable a user to log out and be redirected to a specified page. The URI of the `LogoutController` is typically mapped to the Logout link in the UI. The complete listing of `LogoutController` is as follows:

```
@Controller
public class LogoutController {
  @RequestMapping(value = "/secure/logout",
  method = RequestMethod.GET)
  public String logout(HttpServletRequest request,
  HttpServletResponse response) {
    Authentication auth =
    SecurityContextHolder.getContext()
    .getAuthentication();
    if (auth != null) {
        new SecurityContextLogoutHandler()
       .logout(request, response, auth);
        request.getSession().invalidate();
     }
    return "redirect:/secure/welcome";
   }
 }
```







Things to note are as follows:

- `if (auth != null)`: If there is a valid authentication, then end the session
- `new SecurityContextLogoutHandler().logout(request, response, auth)`: `SecurityContextLogoutHandler` performs a logout by removing the authentication information from `SecurityContextHolder`
- `return "redirect:/secure/welcome"`: Redirects to the secure welcome page

## Summary

In this chapter, we discussed the basics of developing web applications with Spring MVC. We also discussed implementing exception handling, internationalization and securing our applications with Spring Security.

Spring MVC can also be used to build REST services. We will discuss that and more related to REST services in the subsequent chapters.

In the next chapter, we will shift our attention toward microservices. We will try to understand why the world is looking keenly at microservices. We will also explore the importance of applications being Cloud-Native.







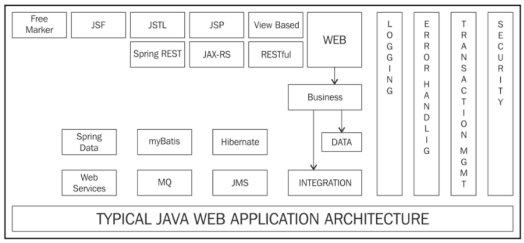


- What are the challenges with microservices?
- What are the good practices that help in deploying microservices to the Cloud?
- What are the Spring projects that help us in developing microservices and Cloud-Native applications?

Native applications?

## Typical web application architecture with Spring

Spring has been the framework of choice to wire Java Enterprise applications during the last decade and half. Applications used a layered architecture with all cross-cutting concerns being managed using aspect-oriented programming. The following diagram shows a typical architecture for a web application developed with Spring:




The typical layers in such an application are listed here. We will list cross-cutting concerns as a separate layer, though in reality, they are applicable across all layers:

- **Web layer**: This is typically responsible for controlling the web application flow (controller and/or Front Controller) and rendering the view.
- **Business layer**: This is where all your business logic is written. Most applications have transaction management starting from the business layer.
- **Data layer**: It is also responsible for talking to the database. This is responsible for persisting/retrieving data in Java objects to the tables in the database.







- **Integration layer**: Applications talk to other applications, either over queues or by invoking web services. The integration layer establishes such connections with other applications.
- **Cross-cutting concerns**: These are concerns across different layers--logging, security, transaction management, and so on. Since Spring IoC container manages the beans, it can weave these concerns around the beans through **Aspect-oriented programming** (**AOP**).

Let's discuss each of the layers and the frameworks used in more detail.

## Web layer

Web layer is dependent on how you would want to expose the business logic to the end user. Is it a web application? Or are you exposing RESTful web services?

### Web application - rendering an HTML View

These web applications use a web MVC framework such as Spring MVC or Struts. The View can be rendered using JSP, JSF, or template-based frameworks such as Freemarker.

### RESTful services

There are two typical approaches used to develop RESTful web services:

- **JAX-RS**: The Java API for REST services. This is a standard from the Java EE specification. Jersey is the reference implementation.
- **Spring MVC or Spring REST**: Restful services can also be developed with Spring MVC.

Spring MVC does not implement JAX-RS so, the choice is tricky. JAX-RS is a Java EE standard. But Spring MVC is more innovative and more likely to help you build new features faster.

## Business layer

The business layer typically contains all the business logic in an application. Spring Framework is used in this layer to wire beans together.







This is also the layer where the boundary of transaction management begins. Transaction management can be implemented using Spring AOP or AspectJ. A decade back, **Enterprise Java Beans** (**EJB**) were the most popular approach to implement your business layer. With its lightweight nature, Spring is now the framework of choice for the business layer.

EJB3 is much simpler than EJB2. However, EJB3 is finding it difficult to recover the ground lost to Spring.

## Data layer

Most applications talk to a database. The data layer is responsible for storing data from your Java objects to your database and vice versa. The following are the most popular approaches to building data layers:

- **JPA**: The **Java Persistence API** helps you to map Java objects (POJOs) to your database tables. Hibernate is the most popular implementation for JPA. JPA is typically preferred for all transactional applications. JPA is not the best choice for batch and reporting applications.
- **MyBatis**: MyBatis (previously, iBatis) is a simple data-mapping framework. As its website (`http://www.mybatis.org/mybatis-3/`) says, *MyBatis is a first class persistence framework with support for custom SQL, stored procedures and advanced mappings. MyBatis eliminates almost all of the JDBC code and manual setting of parameters and retrieval of results*. MyBatis can be considered for batch and reporting applications where SQLs and stored procedures are more typically used.
- **Spring JDBC**: JDBC and Spring JDBC are not that commonly used anymore.

We will discuss in detail the advantages and disadvantages of JDBC, Spring JDBC, MyBatis

and JPA in `Chapter 8`, *Spring Data*.

## Integration layer

The integration layer is typically where we talk to other applications. There might be other applications exposing SOAP or RESTful services over HTTP (the web) or MQ:

- Spring JMS is typically used to send or receive messages on queues or service buses.
- Spring MVC RestTemplate can be used to invoke RESTful services.
- Spring WS can be used to invoke SOAP-based web services.









# Problems solved by Spring

Spring is the framework of choice to wire Enterprise Java applications. It has solved a number of problems that Enterprise Java applications have faced since the complexity associated with EJB2. A few of them are listed as follows:

- Loose coupling and testability
- Plumbing code
- Lightweight architecture
- Architectural flexibility
- Simplified implementation of cross-cutting concerns
- Best design patterns for free

## Loose coupling and testability

Through dependency injection, Spring brings loose coupling between classes. While loose coupling is beneficial to application maintainability in the long run, the first benefits are realized with the testability that it brings in.

Testability was not a forte of Java EE (or J2EE, as it was called then) before Spring. The only way to test EJB2 applications was to run them in the container. Unit testing them was incredibly difficult.

That's exactly the problem Spring Framework set out to solve. As we saw in the earlier chapters, if objects are wired using Spring, writing unit tests becomes easier. We can easily stub or mock dependencies and wire them into objects.

## Plumbing code

The developer of the late 1990s and early-to mid-2000s will be familiar with the amount of plumbing code that had to be written to execute a simple query through JDBC and populate the result into a Java object. You had to perform a **Java Naming and Directory Interface** (**JNDI**) lookup, get a connection, and populate the results. This resulted in duplicate code. Usually, problems were repeated with exception handling code in every method. And this problem is not limited to JDBC.

One of these problems Spring Framework solved was by eliminating all the plumbing code. With Spring JDBC, Spring JMS, and other abstractions, the developers could focus on writing business logic. Spring framework took care of the nitty-gritty.







## Lightweight architecture

Using EJBs made the applications complex, and not all applications needed that complexity. Spring provided a simplified, lightweight way of developing applications. If distribution was needed, it could be added later.

## Architecture flexibility

Spring Framework is used to wire objects across an application in different layers. In spite of its ever-looming presence, Spring Framework did not restrict the flexibility or choice of frameworks that application architects and developers had. A couple of examples are listed as follows:

- Spring Framework provided great flexibility in the web layer. If you wanted to use Struts or Struts 2 instead of Spring MVC, it was configurable. You had the choice of integrating with a wider range of view and template frameworks.
- Another good example is the data layer, where you had possibilities to connect with JPA, JDBC, and mapping frameworks, such as MyBatis.

## Simplified implementation of cross-cutting concerns

When Spring Framework is used to manage beans, the Spring IoC container manages the life cycle--creation, use, auto-wiring, and destruction--of the beans. It makes it easier to weave an additional functionality--such as the cross-cutting concerns--around the beans.

## Design patterns for free

Spring Framework encourages the use of a number of design patterns by default. A few examples are as follows:

- **Dependency Injection or Inversion of Controller**: This is the fundamental design pattern Spring Framework is built to enable. It enables loose coupling and testability.
- **Singleton**: All Spring beans are singletons by default.
- **Factory Pattern**: Using the bean factory to instantiate beans is a good example of the factory pattern.

- **Front Controller**: Spring MVC uses DispatcherServlet as the Front Controller. So we use the Front Controller pattern when we develop applications with Spring MVC.
- **Template Method**: Helps us avoid boilerplate code. Many Spring-based classes--JdbcTemplate and JmsTemplate--are implementations of this pattern.

## Application development goals

Before we move on to the concepts of REST services, microservices, and Cloud-Native applications, let's take some time to understand the common goals we have when we develop applications. Understanding these goals will help us understand why applications are moving toward the microservices architecture.

First of all, we should remember that the software industry is still a relatively young industry. One thing that's been a constant in my decade and a half experience with developing, designing, and architecting software is that things change. The requirements of today are not the requirements of tomorrow. Technology today is not the technology we will use tomorrow. While we can try predicting what happens in the future, we are often wrong.

One of the things we did during the initial decades of software development was build software systems for the future. The design and architecture were made complex in preparation for future requirements.

During the last decade, with **agile** and **extreme programming**, the focus shifted to being **lean** and building good enough systems, adhering to basic principles of design. The focus shifted to evolutionary design. The thought process is this: *If a system has good design for today's needs, and is continuously evolving and has good tests, it can easily be refactored to meet tomorrow's needs*.

While we do not know where we are heading, we do know that a big chunk of our goals when developing applications have not changed.

The key goals of software development, for a large number of applications, can be described with the statement *speed and safety at scale*.

We will discuss each of these in elements in the next section.







## Speed

The speed of delivering new requirements and innovations is increasingly becoming a key differentiator. It is not sufficient to develop (code and test) fast. It is important to deliver (to production) quickly. It is now common knowledge that the best software organizations in the world deliver software to production multiple times every day.

The technology and business landscape is in a constant flux, and is constantly evolving. The key question is "How fast can an application adapt to these changes?". Some of the important changes in the technology and business landscape are highlighted here:

- New programming languages
    - Go
    - Scala
    - Closure
- New programming paradigms
    - Functional programming
    - Reactive programming
- New frameworks
- New tools
    - Development
    - Code quality
    - Automation testing
    - Deployment
    - Containerizations
- New processes and practices
    - Agile
    - Test-driven development
    - Behavior-driven development
    - Continuous integration
    - Continuous delivery
    - DevOps
- New devices and opportunities
    - Mobile
    - Cloud









## Performance

If a web application does not respond within a couple of seconds, there is a very high chance that the user of your application will be disappointed. Performance usually refers to the ability of a system to provide an agreed-upon response time for a defined number of users.

## High resilience

As applications become distributed, the probability of failures increases. How does the application react in the case of localized failures or disruptions? Can it provide basic operations without completely crashing?

This behavior of an application to provide the bare minimum service levels in case of unexpected failures is called resilience.

As more and more applications move towards the Cloud, the resilience of applications becomes important.

We will discuss building highly resilient microservices using *Spring Cloud and Spring Data Flow* in `Chapter 9`, *Spring Cloud* and `Chapter 10`, *Spring Cloud Data Flow*.

## Scalability

Scalability is a measure of how an application would react when the resources at its disposal are scaled up. If an application supports 10,000 users with a given infrastructure, can it support at least 20,000 users with double the infrastructure?

If a web application does not respond within a couple of seconds, there is a very high chance that the user of your application will be disappointed. Performance usually refers to the ability of a system to provide an agreed-upon response time for a defined number of users.

In the world of Cloud, the scalability of applications becomes even more important. It's difficult to guess how successful a startup might be. Twitter or Facebook might not have expected such success when they were incubated. Their success, for large measure, depends on how they were able to adapt to a multi-fold increase in their user base without affecting the performance.

We will discuss building highly scalable microservices using Spring Cloud and Spring Data Flow in `Chapter 9`, *Spring Cloud* and `Chapter 10`, *Spring Cloud Data Flow*.







# Challenges with monolithic applications

Over the last few years, in parallel to working with several small applications, I had the opportunity to work on four different monolithic applications in varied domains--insurance, banking, and health care. All these applications had very similar challenges. In this section, we will start with looking at the characteristics of monoliths and then look at the challenges they bring in.

First of all: What is a monolith? An application with a lot of code--may be greater than 100K lines of code? Yeah.

For me, monoliths are those applications for which getting a release out to production is a big challenge. Applications that fall into this category have a number of user requirements that are immediately needed, but these applications are able to do new feature releases once every few months. Some of these applications even do feature releases once a quarter or sometimes even as less as twice a year.

Typically, all monolithic applications have these characteristics:

- **Large size**: Most of these monolithic applications have more than 100K lines of code. Some have codebases with more than a million lines of code.
- **Large teams**: The team size could vary from 20 to 300.
- **Multiple ways of doing the same thing**: Since the team is huge, there is a communication gap. This results in multiple solutions for the same problem in different parts of the application.
- **Lack of automation testing**: Most of these applications have very few unit tests and a complete lack of integration tests. These applications have great dependence on manual testing.

Because of these characteristics, there are a number of challenges faced by these monolithic applications.

## Long release cycles

Making a code change in one part of the monolith may impact some other part of the monolith. Most code changes will need a complete regression cycle. This results in long release cycles.

Because there is a lack of automation testing, these applications depend on manual testing to find defects. Taking the functionality live is a major challenge.







## Difficult to scale

Typically, most monolithic applications are not Cloud-Native, which means that they are not easy to deploy on the Cloud. They depend on manual installation and manual configuration. There is typically a lot of work put in by the operations team before a new application instance is added to the cluster. This makes scaling up and down a big challenge.

The other important challenge is large databases. Typically, monolithic applications have databases running into **terabytes** (**TB**). The database becomes the bottleneck when scaling up.

## Adapting new technologies

Most monolithic applications use old technologies. Adding a new technology to the monolith only makes it more complex to maintain. Architects and developers are reluctant to bring in any new technologies.

## Adapting new methodologies

New methodologies such as **agile** need small (four-seven team members), independent teams. The big questions with monolith are these: How do we prevent teams from stepping on each other's toes? How do we create islands that enable teams to work independently? This is a difficult challenge to solve.

## Adapting modern development practices

Modern development practices such as **Test-Driven Development** (**TDD**), **Behavior-Driven Development** (**BDD**) need loosely coupled, testable architecture. If the monolithic application has tightly coupled layers and frameworks, it is difficult to unit test. It makes adapting modern development practices challenging.

# Understanding microservices

The challenges with monolithic applications lead to organizations searching for the silver bullet. How will we be able to make more features live more often?

Many organizations have tried different architectures and practices to find a solution.







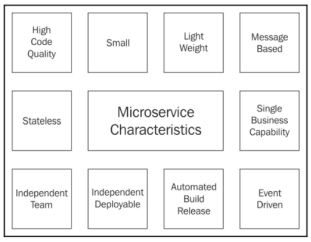


A few important things to note are as follows:

- Modules are identified based on business capabilities. What functionality is the module providing?
- Each module is independently deployable. In the following example, modules 1, 2, and 3 are separate deployable units. If there is a change in the business functionality of module 3, we can individually build and deploy module 3.

## Microservice characteristics

In the previous section, we looked at an example of the microservice architecture. An evaluation of experiences at organizations successful at adapting the microservices architectural style reveals that there are a few characteristics that are shared by teams and architectures. Let's look at some of them:




### Small and lightweight microservices

A good microservice provides a business capability. Ideally, microservices should follow the **single responsibility principle**. Because of this, microservices are generally small in size. Typically, a rule of thumb I use is that it should be possible to build and deploy a microservice within 5 minutes. If the building and deployment takes any longer, it is likely that you are building a larger than recommended microservice.







Some examples of small and lightweight microservices are as follows:

- Product recommendation service
- Email notification service
- Shopping cart service

### Interoperability with message-based communication

The key focus of microservices is on interoperability--communication between systems using diverse technologies. The best way to achieve interoperability is using message-based communication.

### Capability-aligned microservices

It is essential that microservices have a clear boundary. Typically, every microservice has a single identified business capability that it delivers well. Teams have found success adapting the *Bounded Context* concept proposed in the book *Domain-Driven Design* by Eric J Evans.

Essentially, for large systems, it is very difficult to create one domain model. Evans talks about splitting the system into different bounded contexts. Identifying the right bounded contexts is the key to success with microservice architecture.

### Independently deployable units

Each microservice can be individually built and deployed. In the example discussed earlier, modules 1, 2, and 3 can each be independently built and deployed.

### Stateless

An ideal microservice does not have a state. It does not store any information between requests. All the information needed to create a response is present in the request.

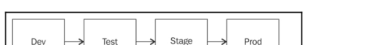


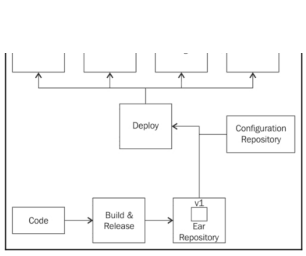




### Automated build and release process

Microservices have automated build and release processes. Take a look at the following figure. It shows a simple build and release process for a microservice:






When a microservice is built and released, a version of the microservice is stored in the repository. The deploy tool has the capability of picking the right version of microservice from the repository, matching it with the configuration needed for the specific environment (from the configuration repository), and deploying the microservice to a specific environment.

Some teams take it a step further and combine the microservice package with the underlying infrastructure needed to run the microservice. The deploy tool will replicate this image and match it with an environment-specific configuration to create an environment.

## Event-driven architecture

Microservices are typically built with event-driven architecture. Let's consider a simple example. Whenever a new customer is registered, there are three things that need to be performed:

- Store customer information to the database
- Mail a welcome kit
- Send an email notification







Let's look at two different approaches to design this.

### Approach 1 - sequential approach

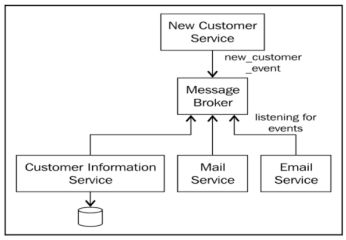
Let's consider three services--`CustomerInformationService`, `MailService`, and `EmailService`, which can provide the capabilities listed earlier. We can create `NewCustomerService` with the following steps:

1. Call `CustomerInformationService` to save customer information to the database.
2. Call `MailService` to mail the welcome kit.
3. Call `EmailService` to send the e-mail notification.

`NewCustomerService` becomes the central place for all business logic. Imagine if we have to do more things when a new customer is created. All that logic would start accumulating and bloating up `NewCustomerService`.

### Approach 2 - event-driven approach

In this approach, we use a message broker. `NewCustomerService` will create a new event and post it to the message broker. The following figure shows a high-level representation:










The three services--`CustomerInformationService`, `MailService`, and `EmailService`--will be listening on the message broker for new events. When they see the new customer event, they process it and execute the functionality of that specific service.

The key advantage of the event-driven approach is that there is no centralized magnet for all the business logic. Adding a new functionality is easier. We can create a new service to listen for the event on the message broker. Another important thing to note is that we don't need to make changes to any of the existing services.

## Independent teams

The team developing a microservice is typically independent. It contains all the skills needed to develop, test, and deploy a microservice. It is also responsible for supporting the microservice in production.

# Microservice advantages

Microservices have several advantages. They help in keeping up with technology and getting solutions to your customers faster.

## Faster time to market

The faster time to market is one of the key factors in determining the success of an organization.

Microservices architecture involves creating small, independently deployable components. Microservice enhancements are easier and less brittle because each microservice focuses on a single business capability. All the steps in the process--building, releasing, deployment, testing, configuration management, and monitoring--are automated. Since the responsibility of a microservice is bounded, it is possible to write great automation unit and integration

tests.

All these factors result in applications being able to react faster to customer needs.

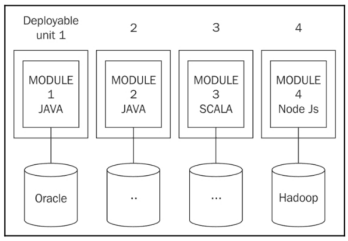






## Technology evolution

There are new languages, frameworks, practices, and automation possibilities emerging every day. It is important that the application architectures allow flexibility to adapt to emerging possibilities. The following figure shows how different services are developed in different technologies:




The microservice architecture involves creating small services. Within some boundaries, most organizations give the individual teams the of technology to make some of the technology decisions. This allows teams to experiment with new technologies and innovate faster. This helps applications adapt and stay in tune with the evolution of technology.

## Availability and scaling

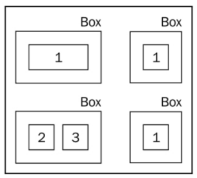
The load on different parts of the application is typically very different. For example, in the case of a flight booking application, a customer usually searches multiple times before making a decision on whether to book a flight. The load on the search module would typically be many times more than the load on the booking module. The microservices architecture provides the flexibility of setting up multiple instances of the search service with few instances of the booking service.







The following figure shows how we can scale up specific microservices based on the load:




Microservices **2** and **3** share a single box (the deployment environment). Microservice **1**, which has more load, is deployed into multiple boxes.

Another example is the need for start-ups. When a start-up begins its operations, they are typically unaware of the extent to which they might grow. What happens if the demand for applications grows very fast? If they adapt the microservice architecture, it enables them to scale better when the need arises.

## Team dynamics

Development methodologies such as agile advocate small, independent teams. Since microservices are small, it is possible to build small teams around them. Teams are cross-functional, with end-to-end ownership of specific microservices.

Microservice architecture fits in very well with agile and other modern development methodologies.

# Microservice challenges

Microservice architecture has significant advantages. However, there are significant challenges too. Deciding the boundaries of microservices is a challenging but important decision. Since microservices are small, and there would be hundreds of microservices in a large enterprise, having great automation and visibility is critical.







## Increased need for automation

With microservice architecture, you are splitting up a large application into multiple

microservices, so the number of builds, releases, and deployments increases multifold. It would be very inefficient to have manual processes for these steps.

Test automation is critical to enable a faster time to market. Teams should be focused on identifying automation possibilities as they emerge.

## Defining the boundaries of subsystems

Microservices should be intelligent. They are not weak CRUD services. They should model the business capability of the system. They own all the business logic in a bounded context. Having said this, microservices should not be large. Deciding the boundaries of microservices is a challenge. Finding the right boundaries might be difficult on the first go. It is important that as a team gains more knowledge about the business context, the knowledge flows into the architecture and new boundaries are determined. Generally, finding the right boundaries for microservices is an evolutionary process.

A couple of important points to note are as follows:

- Loose coupling and high cohesion are fundamental to any programming and architectural decisions. When a system is loosely coupled, changes in one part should not require a change in other parts.
- Bounded contexts represent autonomous business modules representing specific business capabilities.

*As Sam Newman says in the book Building Microservices--"Specific responsibility enforced by explicit boundaries". Always think, "What capabilities are we providing to the rest of the domain?".*

## Visibility and monitoring

With microservices, one application is split into several microservices. To conquer the complexity associated with multiple microservices and asynchronous event-based collaboration, it is important to have great visibility.

Ensuring high availability means each microservice should be monitored. Automated health management of the microservices becomes important.







Debugging problems needs insights into what's happening behind multiple microservices. Centralized logging with aggregation of logs and metrics from different microservices is typically used. Mechanisms such as correlation IDs need to be used to isolate and debug issues.

## Fault tolerance

Let's say we are building a shopping application. What happens if the recommendation microservice is down? How does the application react? Does it completely crash? Or will it let the customer shop? These kinds of situations happen more often as we adapt the microservices architecture.

As we make the services small, the chance that a service is down increases. How the application reacts to these situations becomes an important question. In the earlier example, a fault-tolerant application would show some default recommendations while letting the customer shop.

As we move into microservices architecture, applications should be more fault tolerant. Applications should be able to provide toned-down behavior when services are down.

## Eventual consistency

It is important to have a degree of consistency between microservices in an organization. Consistency between microservices enables similar development, testing, release, deployment, and operational processes across the organization. This enables different developers and testers to be productive when they move across teams. It is important to be not very rigid and have a degree of flexibility within limits so as to not stifle innovation.

### Shared capabilities (enterprise level)

Let's look at a few capabilities that have to be standardized at an enterprise level.

- **Hardware**: What hardware do we use? Do we use Cloud?
- **Code management**: What version control system do we use? What are our practices in branching and committing code?
- **Build and deployment**: How do we build? What tools do we use to automate deployment?
- **Data store**: What kind of data stores do we use?









Cloud-Native applications are those that can easily be deployed on the Cloud. These applications share a few common characteristics. We will begin by looking at the Twelve-Factor App--a combination of common patterns among Cloud-Native applications.

# Twelve-Factor App

The Twelve-Factor App evolved from the experiences of engineers at Heroku. It is a list of patterns that are used in Cloud-Native application architectures.

It is important to note that an app here refers to a single deployable unit. Essentially, every microservice is an app (because each microservice is independently deployable).

## Maintain one code base

Each app has one code base in revision control. There can be multiple environments where the app can be deployed. However, all these environments use code from a single codebase. An example antipattern is building a deployable from multiple codebases.

## Dependencies

All dependencies must be explicitly declared and isolated. Typical Java applications use

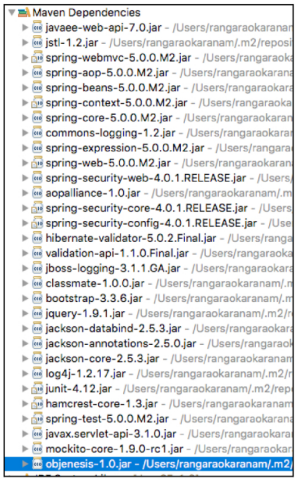
build management tools such as Maven and Gradle to isolate and track dependencies.







The following figure shows typical Java applications managing dependencies using Maven:

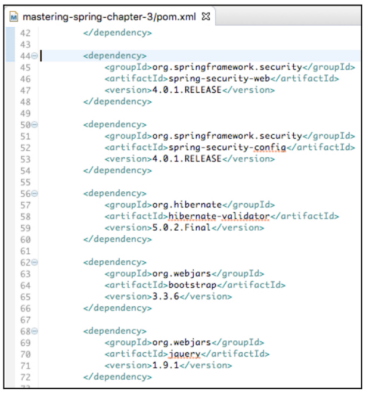









The following figure shows `pom.xml`, where the dependencies are managed for a Java application:




## Config

All applications have configuration that varies from one environment to another. Configuration is found at multiple locations; application code, property files, databases, environment variables, JNDI, and system variables are a few examples.

### Disposability

A Twelve-Factor App should promote elastic scaling. Hence, they should be disposable. They can be started and stopped when needed.

A Twelve-Factor App should do the following:

- Have minimum startup time. A long startup time means a long delay before an application can take requests.
- Shut down gracefully.
- Handle hardware failures gracefully.

### Environment parity

All the environments--development, test, staging, and production--should be similar. They should use the same processes and tools. With continuous deployment, they should have very frequently have similar code. This makes finding and fixing problems easier.

### Logs as event streams

Visibility is critical to a Twelve-Factor App. Since applications are deployed on the Cloud and are automatically scaled, it is important that you have a centralized view of what's happening across different instances of the applications.

Treating all logs as stream enables the routing of the log stream to different destinations for viewing and archival purposes. This stream can be used to debug issues, perform analytics, and create alerting systems based on error patterns.

### No distinction of admin processes

Twelve-Factor Apps treat administrative tasks (migrations, scripts) similar to normal application processes.

## Spring projects

As the world moves toward Cloud-Native applications and microservices, Spring projects are not far behind. There are a number of new Spring projects--Spring Boot, Spring Cloud, among others, that solve the problems of the emerging world.







## Spring Boot

In the era of monoliths, we had the luxury of taking the time to set the frameworks up for an application. However, in the era of microservices, we want to create individual components faster. The Spring Boot project aims to solve this problem.

As the official website highlights, Spring Boot makes it easy to create standalone, production-grade Spring-based applications that you can *just run*. We take an opinionated view of the Spring platform and third-party libraries so that you can get started with minimum fuss.

Spring Boot aims to take an opinionated view--basically making a lot of decisions for us--to developing Spring-based projects.

In the next couple of chapters, we will look at Spring Boot and the different features that enable us to create production-ready applications faster.

## Spring Cloud

Spring Cloud aims to provide solutions to some commonly encountered patterns when building systems on the Cloud:

- **Configuration management**: As we discussed in the Twelve-Factor App section, managing configuration is an important part of developing Cloud-Native applications. Spring Cloud provides a centralized configuration management solution for microservices called Spring Cloud Config.
- **Service discovery**: Service discovery promotes loose coupling between services. Spring Cloud provides integration with popular service discovery options, such as Eureka, ZooKeeper, and Consul.
- **Circuit breakers**: Cloud-Native applications must be fault tolerant. They should be able to handle the failure of backing services gracefully. Circuit breakers play a key role in providing the default minimum service in case of failures. Spring Cloud provides integration with the Netflix Hystrix fault tolerance library.
- **API Gateway**: An API Gateway provides centralized aggregation, routing, and caching services. Spring Cloud provides integration with the API Gateway library Netflix Zuul.







## Summary

In this chapter, we looked at how the world evolved toward microservices and Cloud-Native applications. We understood how Spring Framework and projects are evolving to meet the needs of today's world with projects such as Spring Boot, Spring Cloud, and Spring Data.

In the next chapter, we will start focusing on Spring Boot. We will look at how Spring Boot makes developing microservices easy.

Pages 139 to 145 are not shown in this preview.



## Understanding starter projects

Starters are simplified dependency descriptors customized for different purposes. For example, `spring-boot-starter-web` is the starter for building web application, including RESTful, using Spring MVC. It uses Tomcat as the default embedded container. If I want to develop a web application using Spring MVC, all we would need to do is include `spring-boot-starter-web` in our dependencies, and we get the following automatically pre-configured:

- Spring MVC
- Compatible versions of jackson-databind (for binding) and hibernate-validator (for form validation)
- `spring-boot-starter-tomcat` (starter project for Tomcat)

The following code snippet shows some of the dependencies configured in `spring-boot-starter-web`:

```
<dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-tomcat</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-validator</artifactId>
    </dependency>
    <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-web</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-webmvc</artifactId>
   </dependency>
</dependencies>
```







As we can see in the preceding snippet, when we use `spring-boot-starter-web`, we get a lot of frameworks auto-configured.

For the web application we would like to build, we would also want to do some good unit testing and deploy it on Tomcat. The following snippet shows the different starter dependencies that we would need. We would need to add this to our `pom.xml` file:

```
<dependencies>
  <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>
 <dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-tomcat</artifactId>
   <scope>provided</scope>
 </dependency>
</dependencies>
```

We add three starter projects:

- We've already discussed `spring-boot-starter-web`. It provides us with the frameworks needed to build a web application with Spring MVC.
- The `spring-boot-starter-test` dependency provides the following test frameworks needed for unit testing:
    - **JUnit**: Basic unit test framework
    - **Mockito**: For mocking
    - **Hamcrest**, **AssertJ**: For readable asserts
    - **Spring Test**: A unit testing framework for spring-context based applications
- The `spring-boot-starter-tomcat` dependency is the default for running web applications. We include it for clarity. `spring-boot-starter-tomcat` is the

starter for using Tomcat as the embedded servlet container.







We now have our `pom.xml` file configured with the starter parent and the required starter projects. Let's add `spring-boot-maven-plugin` now, which would enable us to run Spring Boot applications.

## Configuring spring-boot-maven-plugin

When we build applications using Spring Boot, there are a couple of situations that are possible:

- We would want to run the applications in place without building a JAR or a WAR
- We would want to build a JAR and a WAR for later deployment

The `spring-boot-maven-plugin` dependency provides capabilities for both of the preceding situations. The following snippet shows how we can configure `spring-boot-maven-plugin` in an application:

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

The `spring-boot-maven-plugin` dependency provides several goals for a Spring Boot application. The most popular goal is run (this can be executed as `mvn spring-boot:run` on the command prompt from the root folder of the project).

## Creating your first Spring Boot launch class

The following class explains how to create a simple Spring Boot launch class. It uses the static run method from the `SpringApplication` class, as shown in the following code snippet:

```
package com.mastering.spring.springboot;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.
autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
@SpringBootApplication public class Application {
```





Pages 149 to 153 are not shown in this preview.



```
mvcValidator
mvcViewResolver
objectNamingStrategy
autoconfigure.AutoConfigurationPackages
autoconfigure.PropertyPlaceholderAutoConfiguration
autoconfigure.condition.BeanTypeRegistry
autoconfigure.context.ConfigurationPropertiesAutoConfiguration
autoconfigure.info.ProjectInfoAutoConfiguration
autoconfigure.internalCachingMetadataReaderFactory
autoconfigure.jackson.JacksonAutoConfiguration
autoconfigure.jackson.JacksonAutoConfiguration$Jackson2ObjectMapperBuilderC
ustomizerConfiguration
autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMapperBuilderCo
nfiguration
autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMapperConfigura
tion
autoconfigure.jmx.JmxAutoConfiguration
autoconfigure.web.DispatcherServletAutoConfiguration
autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherServletConfi
guration
autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherServletRegis
trationConfiguration
autoconfigure.web.EmbeddedServletContainerAutoConfiguration
autoconfigure.web.EmbeddedServletContainerAutoConfiguration$EmbeddedTomcat
autoconfigure.web.ErrorMvcAutoConfiguration
autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguratio
n
autoconfigure.web.HttpEncodingAutoConfiguration
autoconfigure.web.HttpMessageConvertersAutoConfiguration
autoconfigure.web.HttpMessageConvertersAutoConfiguration$StringHttpMessageC
onverterConfiguration
autoconfigure.web.JacksonHttpMessageConvertersConfiguration
autoconfigure.web.JacksonHttpMessageConvertersConfiguration$MappingJackson2
HttpMessageConverterConfiguration
autoconfigure.web.MultipartAutoConfiguration
autoconfigure.web.ServerPropertiesAutoConfiguration
autoconfigure.web.WebClientAutoConfiguration
autoconfigure.web.WebClientAutoConfiguration$RestTemplateConfiguration
autoconfigure.web.WebMvcAutoConfiguration
autoconfigure.web.WebMvcAutoConfiguration$EnableWebMvcConfiguration
autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter
autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter$Fa
viconConfiguration
autoconfigure.websocket.WebSocketAutoConfiguration
autoconfigure.websocket.WebSocketAutoConfiguration$TomcatWebSocketConfigura
tion
context.properties.ConfigurationPropertiesBindingPostProcessor
context.properties.ConfigurationPropertiesBindingPostProcessor.store
```







```
annotation.ConfigurationClassPostProcessor.enhancedConfigurationProcessor
annotation.ConfigurationClassPostProcessor.importAwareProcessor
```

```
annotation.internalAutowiredAnnotationProcessor
annotation.internalCommonAnnotationProcessor
annotation.internalConfigurationAnnotationProcessor
annotation.internalRequiredAnnotationProcessor
event.internalEventListenerFactory
event.internalEventListenerProcessor
preserveErrorControllerTargetClassPostProcessor
propertySourcesPlaceholderConfigurer
requestContextFilter
requestMappingHandlerAdapter
requestMappingHandlerMapping
resourceHandlerMapping
restTemplateBuilder
serverProperties
simpleControllerHandlerAdapter
spring.http.encoding-autoconfigure.web.HttpEncodingProperties
spring.http.multipart-autoconfigure.web.MultipartProperties
spring.info-autoconfigure.info.ProjectInfoProperties
spring.jackson-autoconfigure.jackson.JacksonProperties
spring.mvc-autoconfigure.web.WebMvcProperties
spring.resources-autoconfigure.web.ResourceProperties
standardJacksonObjectMapperBuilderCustomizer
stringHttpMessageConverter
tomcatEmbeddedServletContainerFactory
viewControllerHandlerMapping
viewResolver
websocketContainerCustomizer
```

Important things to think about are as follows:

- Where are these beans defined?
- How are these beans created?

That's the magic of Spring auto-configuration.

Whenever we add a new dependency to a Spring Boot project, Spring Boot auto-configuration automatically tries to configure the beans based on the dependency.

For example, when we add a dependency in `spring-boot-starter-web`, the following beans are auto-configured:

- `basicErrorController`, `handlerExceptionResolver`: Basic exception handling. Shows a default error page when an exception occurs.
- `beanNameHandlerMapping`: Used to resolve paths to a handler (controller).







- `characterEncodingFilter`: Provides default character encoding UTF-8.
- `dispatcherServlet`: DispatcherServlet is the Front Controller in Spring MVC applications.
- `jacksonObjectMapper`: Translates objects to JSON and JSON to objects in REST services.
- `messageConverters`: The default message converters to convert from objects into XML or JSON and vice versa.
- `multipartResolver`: Provides support to upload files in web applications.
- `mvcValidator`: Supports validation of HTTP requests.
- `viewResolver`: Resolves a logical view name to a physical view.
- `propertySourcesPlaceholderConfigurer`: Supports the externalization of application configuration.
- `requestContextFilter`: Defaults the filter for requests.
- `restTemplateBuilder`: Used to make calls to REST services.
- `tomcatEmbeddedServletContainerFactory`: Tomcat is the default embedded servlet container for Spring Boot-based web applications.

In the next section, let's look at some of the starter projects and the auto-configuration they provide.

## Starter projects

The following table shows some of the important starter projects provided by Spring Boot:

| Starter | Description |
|---|---|
| `spring-boot-starter-web-services` | This is a starter project to develop XML-based web services. |
| `spring-boot-starter-web` | This is a starter project to build Spring MVC-based web applications or RESTful applications. It uses Tomcat as the default embedded servlet container. |
| `spring-boot-starter-activemq` | This supports message-based communication using JMS on ActiveMQ. |
| `spring-boot-starter-integration` | This supports the Spring Integration Framework that provides implementations for Enterprise Integration Patterns. |









```
    @Test
    public void welcome() throws Exception {
      mvc.perform(
       MockMvcRequestBuilders.get("/welcome")
      .accept(MediaType.APPLICATION_JSON))
      .andExpect(status().isOk())
      .andExpect(content().string(
      equalTo("Hello World")));
    }
}
```

In the preceding unit test, we will launch up a Mock MVC instance with `BasicController`. A few quick things to note are as follows:

- `@RunWith(SpringRunner.class)`: SpringRunner is a shortcut to the `SpringJUnit4ClassRunner` annotation. This launches up a simple Spring context for unit testing.
- `@WebMvcTest(BasicController.class)`: This annotation can be used along with SpringRunner to write simple tests for Spring MVC controllers. This will load only the beans annotated with Spring-MVC-related annotations. In this example, we are launching a Web MVC Test context with the class under test being BasicController.
- `@Autowired private MockMvc mvc`: Autowires the MockMvc bean that can be

used to make requests.

- `mvc.perform(MockMvcRequestBuilders.get("/welcome").accept(MediaType.APPLICATION_JSON))`: Performs a request to `/welcome` with the `Accept` header value `application/json`.
- `andExpect(status().isOk())`: Expects that the status of the response is 200 (success).
- `andExpect(content().string(equalTo("Hello World")))`: Expects that the content of the response is equal to `"Hello World"`.

## Integration testing

When we do integration testing, we would want to launch the embedded server with all the controllers and beans that are configured. This code snippet shows how we can create a simple integration test:

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class,
webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class BasicControllerIT {
```







```
  private static final String LOCAL_HOST =
  "http://localhost:";

  @LocalServerPort
  private int port;

  private TestRestTemplate template = new TestRestTemplate();

  @Test
  public void welcome() throws Exception {
    ResponseEntity<String> response = template
   .getForEntity(createURL("/welcome"), String.class);
    assertThat(response.getBody(), equalTo("Hello World"));
   }

  private String createURL(String uri) {
    return LOCAL_HOST + port + uri;
   }
}
```

A few important things to note are as follows:

- `@SpringBootTest(classes = Application.class, webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)`: Provides additional functionality on top of the Spring TestContext. Provides support to configure the port for fully running the container and TestRestTemplate (to execute requests).
- `@LocalServerPort private int port`: `SpringBootTest` would ensure that the port on which the container is running is autowired into the port variable.
- `private String createURL(String uri)`: The method to append the local host URL and port to the URI to create a full URL.
- `private TestRestTemplate template = new TestRestTemplate()`: `TestRestTemplate` is typically used in integration tests. It provides additional functionality on top of RestTemplate, which is especially useful in the integration test context. It does not follow redirects so that we can assert response location.
- `template.getForEntity(createURL("/welcome"), String.class)`: Executes a get request for the given URI.
- `assertThat(response.getBody(), equalTo("Hello World"))`: Asserts that the response body content is `"Hello World"`.







# Simple REST method returning an object

In the previous method, we returned a string. Let's create a method that returns a proper JSON response. Take a look at the following method:

```
@GetMapping("/welcome-with-object")
public WelcomeBean welcomeWithObject() {
  return new WelcomeBean("Hello World");
}
```

This preceding method returns a simple `WelcomeBean` initialized with a message: `"Hello World"`.

## Executing a request

Let's send a test request and see what response we get. The following screenshot shows the output:

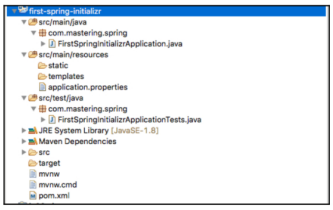

The response for the `http://localhost:8080/welcome-with-object` URL is shown as follows:

```
{"message":"Hello World"}
```

The question that needs to be answered is this: how does the `WelcomeBean` object that we returned get converted into JSON?

Again, it's the magic of Spring Boot auto-configuration. If Jackson is on the classpath of an application, instances of the default object to JSON (and vice versa) converters are auto-configured by Spring Boot.

This will import the project into Eclipse. The following screenshot shows the structure of the project in Eclipse:




Let's look at some of the important files from the generated project.

## pom.xml

The following snippet shows the dependencies that are declared:

```
<dependencies> <dependency> <groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId> </dependency> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
starter-data-jpa</artifactId> </dependency> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
starter-actuator</artifactId> </dependency> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
devtools</artifactId> <scope>runtime</scope> </dependency> <dependency>
<groupId>org.springframework.boot</groupId> <artifactId>spring-boot-
starter-test</artifactId> <scope>test</scope> </dependency> </dependencies>
```

A few other important observations are as follows:

- The packaging for this component is `.jar`
- `org.springframework.boot:spring-boot-starter-parent` is declared as the parent POM
- `<java.version>1.8</java.version>`: The Java version is 1.8
- Spring Boot Maven Plugin (`org.springframework.boot:spring-boot-maven-plugin`) is configured as a plugin







## FirstSpringInitializrApplication.java class

`FirstSpringInitializrApplication.java` is the launcher for Spring Boot:

```
package com.mastering.spring;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure
.SpringBootApplication;

@SpringBootApplication
public class FirstSpringInitializrApplication {
   public static void main(String[] args) {
    SpringApplication.run(FirstSpringInitializrApplication.class,
    args);
  }
}
```

## FirstSpringInitializrApplicationTests class

`FirstSpringInitializrApplicationTests` contains the basic context that can be used to start writing the tests as we start developing the application:

```
package com.mastering.spring;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class FirstSpringInitializrApplicationTests {

  @Test
  public void contextLoads() {
  }
}
```

# A quick peek into auto-configuration

Auto-configuration is one of the most important features of Spring Boot. In this section, we will take a quick peek behind the scenes to understand how Spring Boot auto-configuration works.

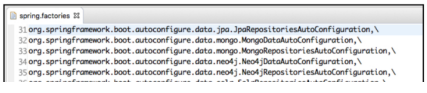






Most of the Spring Boot auto-configuration magic comes from `spring-boot-autoconfigure-{version}.jar`. When we start any Spring Boot applications, a number of beans get auto-configured. How does this happen?

The following screenshot shows an extract from `spring.factories` from `spring-boot-autoconfigure-{version}.jar`. We have filtered out some of the configuration in the interest of space:



```
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.hornetq.HornetQAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
```

The preceding list of auto-configuration classes is run whenever a Spring Boot application is launched. Let's take a quick look at one of them:

`org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration`.









As we can see in the preceding two examples, Spring Boot provides good default exception handling. In the next section, we will focus on understanding how the application reacts to custom exceptions.

## Throwing a custom exception

Let's create a custom exception and throw it from a service. Take a look at the following code:

```
public class TodoNotFoundException extends RuntimeException {
  public TodoNotFoundException(String msg) {
    super(msg);
  }
}
```

It's a very simple piece of code that defines `TodoNotFoundException`.

Now let's enhance our `TodoController` class to throw `TodoNotFoundException` when a `todo` with a given ID is not found:

```
@GetMapping(path = "/users/{name}/todos/{id}")
public Todo retrieveTodo(@PathVariable String name,
@PathVariable int id) {
  Todo todo = todoService.retrieveTodo(id);
  if (todo == null) {
    throw new TodoNotFoundException("Todo Not Found");
   }

 return todo;
}
```

If `todoService` returns a null `todo`, we throw `TodoNotFoundException`.

When we execute the service with a `GET` request to a nonexistent `todo` (`http://localhost:8080/users/Jack/todos/222`), we get the response shown in the following code snippet:

```
{
  "timestamp": 1484029048788,
  "status": 500,
  "error": "Internal Server Error",
  "exception":
  "com.mastering.spring.springboot.bean.TodoNotFoundException",
  "message": "Todo Not Found",
  "path": "/users/Jack/todos/222"
}
```







As we can see, a clear exception response is sent back to the service consumer. However, there is one thing that can be improved further--the response status. When a resource is not found, it is recommended that you return a `404 - Resource Not Found` status. We will look at how to customize the response status in the next example.

## Customizing the exception message

Let's look at how to customize the preceding exception and return the proper response status with a customized message.

Let's create a bean to define the structure of our custom exception message:

```
public class ExceptionResponse {
  private Date timestamp = new Date();
  private String message;
  private String details;

  public ExceptionResponse(String message, String details) {
    super();
    this.message = message;
    this.details = details;
   }

  public Date getTimestamp() {
    return timestamp;
  }

  public String getMessage() {
    return message;
  }

  public String getDetails() {
    return details;
  }
}
```

We have created a simple exception response bean with an auto-populated timestamp with

a few additional properties namely messages and details.







When `TodoNotFoundException` is thrown, we would want to return a response using the `ExceptionResponse` bean. The following code shows how we can create a global exception handling for `TodoNotFoundException.class`:

```
@ControllerAdvice
@RestController
public class RestResponseEntityExceptionHandler
  extends  ResponseEntityExceptionHandler
  {
    @ExceptionHandler(TodoNotFoundException.class)
    public final ResponseEntity<ExceptionResponse>
    todoNotFound(TodoNotFoundException ex) {
       ExceptionResponse exceptionResponse =
       new ExceptionResponse(  ex.getMessage(),
       "Any details you would want to add");
       return new ResponseEntity<ExceptionResponse>
       (exceptionResponse, new HttpHeaders(),
       HttpStatus.NOT_FOUND);
     }
 }
```

Some important things to note are as follows:

- `RestResponseEntityExceptionHandler extends ResponseEntityExceptionHandler`: We are extending `ResponseEntityExceptionHandler`, which is the base class provided by Spring MVC for centralised exception handling `ControllerAdvice` classes.
- `@ExceptionHandler(TodoNotFoundException.class)`: This defines that the method to follow will handle the specific exception `TodoNotFoundException.class`. Any other exceptions for which custom exception handling is not defined will follow the default exception handling provided by Spring Boot.
- `ExceptionResponse exceptionResponse = new ExceptionResponse(ex.getMessage(), "Any details you would want to add")`: This creates a custom exception response.
- `new ResponseEntity<ExceptionResponse>(exceptionResponse,new HttpHeaders(), HttpStatus.NOT_FOUND)`: This is the definition to return a `404 Resource Not Found` response with the custom exception defined earlier.





Pages 196 to 197 are not shown in this preview.



> *A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations. A REST API should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience (i.e., expected to be understood by any client that might use the API). From that point on, all application state transitions must be driven by client selection of server-provided choices that are present in the received representations or implied by the user's manipulation of those representations. The transitions may be determined (or limited by) the client's knowledge of media types and resource communication mechanisms, both of which may be improved on-the-fly (e.g., code-on-demand).*

An example response with HATEOAS link is shown here. This is the response to the `/todos` request in order to retrieve all todos:

```
{
  "_embedded" : {
    "todos" : [ {
      "user" : "Jill",
      "desc" : "Learn Hibernate",
      "done" : false,
     "_links" : {
       "self" : {
              "href" : "http://localhost:8080/todos/1"
             },
       "todo" : {
              "href" : "http://localhost:8080/todos/1"
               }
        }
 } ]
},
 "_links" : {
 "self" : {
          "href" : "http://localhost:8080/todos"
          },
 "profile" : {
          "href" : "http://localhost:8080/profile/todos"
          },
 "search" : {
          "href" : "http://localhost:8080/todos/search"
          }
  },
 }
```

The preceding response includes links to the following:

- Specific `todos` (`http://localhost:8080/todos/1`)
- Search resource (`http://localhost:8080/todos/search`)

If the service consumer wants to do a search, it has the option of taking the search URL from the response and sending the search request to it. This would reduce coupling between the service provider and the service consumer.

## Sending HATEOAS links in response

Now that we understand what HATEOAS is, let's look at how we can send links related to a resource in the response.

## Spring Boot starter HATEOAS

Spring Boot has a specific starter for HATEOAS called `spring-boot-starter-hateoas`. We need to add it to the `pom.xml` file.

The following code snippet shows the dependency block:

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

One of the important dependencies of `spring-boot-starter-hateoas` is `spring-hateoas`, which provides the HATEOAS features:

```
<dependency>
  <groupId>org.springframework.hateoas</groupId>
  <artifactId>spring-hateoas</artifactId>
</dependency>
```

Let's enhance the `retrieveTodo` resource (`/users/{name}/todos/{id}`) to return a link to retrieve all `todos` (`/users/{name}/todos`) in the response:

```
@GetMapping(path = "/users/{name}/todos/{id}")
public Resource<Todo> retrieveTodo(
@PathVariable String name, @PathVariable int id) {
Todo todo = todoService.retrieveTodo(id);
  if (todo == null) {
        throw new TodoNotFoundException("Todo Not Found");
```







```
    }

 Resource<Todo> todoResource = new Resource<Todo>(todo);
 ControllerLinkBuilder linkTo =
 linkTo(methodOn(this.getClass()).retrieveTodos(name));
 todoResource.add(linkTo.withRel("parent"));

 return todoResource;
}
```

Some important points to note are as follows:

- `ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveTodos(name))`: We want to get a link to the `retrieveTodos` method on the current class
- `linkTo.withRel("parent")`: Relationship with the current resource is parent

The following snippet shows the response when a `GET` request is sent to `http://localhost:8080/users/Jack/todos/1`:

```
{
  "id": 1,
  "user": "Jack",
  "desc": "Learn Spring MVC",
  "targetDate": 1484038262110,
  "done": false,
  "_links": {
            "parent": {
            "href": "http://localhost:8080/users/Jack/todos"
            }
    }
}
```

The `_links` section will contain all the links. Currently, we have one link with the relation parent and `href` as `http://localhost:8080/users/Jack/todos`.

If you have problems executing the preceding request, try executing using an Accept header--`application/json`.

HATEOAS is not something that is commonly used in most of the resources today. However, it has the potential to be really useful in reducing the coupling between the service provider and the consumer.







# Validation

A good service always validates data before processing it. In this section, we will look at the Bean Validation API and use its reference implementation to implement validation in our services.

The Bean Validation API provides a number of annotations that can be used to validate beans. The *JSR 349* specification defines Bean Validation API 1.1. Hibernate-validator is the reference implementation. Both are already defined as dependencies in the `spring-boot-web-starter` project:

- `hibernate-validator-5.2.4.Final.jar`
- `validation-api-1.1.0.Final.jar`

We will create a simple validation for the createTodo service method.

Creating validations involves two steps:

1. Enabling validation on the controller method.
2. Adding validations on the bean.

## Enabling validation on the controller method

It's very simple to enable validation on the controller method. The following snippet shows

an example:

```
@RequestMapping(method = RequestMethod.POST,
path = "/users/{name}/todos")
ResponseEntity<?> add(@PathVariable String name
@Valid @RequestBody Todo todo) {
```

The `@Valid(package javax.validation)` annotation is used to mark a parameter for validation. Any validation that is defined in the `Todo` bean is executed before the `add` method is executed.

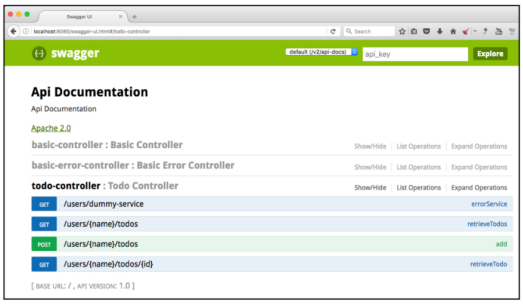




Pages 202 to 208 are not shown in this preview.



The following screenshot shows the list of controller-exposing services. When we click on any controller, it expands to show the list of request methods and URIs each controller supports:

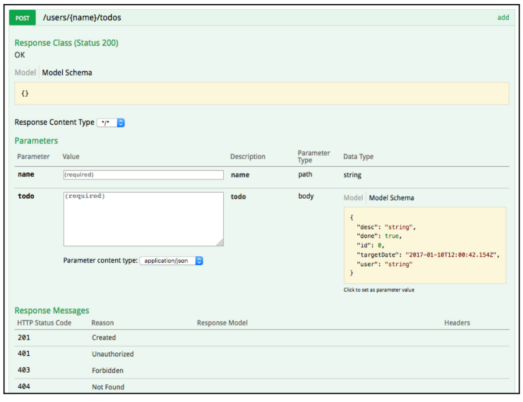









The following screenshot shows the details for the POST service to create a `todo` for the user in Swagger UI:




Some important things to note are as follows:

- **Parameters** show all the important parameters including the request body
- The **Parameter Type** body (for the `todo` parameter) shows the expected structure for the body of the request
- The **Response Messages** sections show different HTTP status codes returned by the service

The Swagger UI provides an excellent way to expose service definitions for your API without a lot of additional effort.

## Customizing Swagger documentation using annotations

The Swagger UI also provides annotations to further customize your documentation.

Listed here is some of the documentation to retrieve the `todos` service:

```
"/users/{name}/todos": {
  "get": {
  "tags": [
         "todo-controller"
        ],
  "summary": "retrieveTodos",
  "operationId": "retrieveTodosUsingGET",
  "consumes": [
           "application/json"
          ],
  "produces": [
            "*/*"
           ],
```

As you can see, the documentation generated is very raw. There are a number of things we can improve in the documentation to describe the services better. Here are a couple of examples:

- Provide a better summary
- Add application/JSON to produces

Swagger provides annotations we can add to our RESTful services in order to customize the documentation. Let's add a few annotations to the controller in order to improve the documentation:

```
@ApiOperation(
  value = "Retrieve all todos for a user by passing in his name",
  notes = "A list of matching todos is returned. Current pagination
  is not supported.",
  response = Todo.class,
  responseContainer = "List",
  produces = "application/json")
  @GetMapping("/users/{name}/todos")
  public List<Todo> retrieveTodos(@PathVariable String name) {
    return todoService.retrieveTodos(name);
 }
```









An example is shown in the following code snippet:

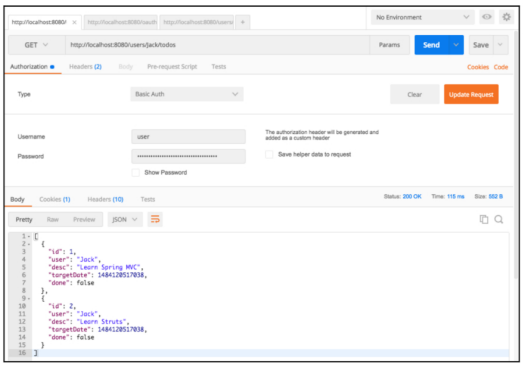
```
2017-01-11 13:11:58.696 INFO 3888 --- [ restartedMain]
b.a.s.AuthenticationManagerConfiguration :

Using default security password: 3fb5564a-ce53-4138-9911-8ade17b2f478

2017-01-11 13:11:58.771 INFO 3888 --- [ restartedMain]
o.s.s.web.DefaultSecurityFilterChain : Creating filter chain: Ant
[pattern='/css/**'], []
```

Underlined in the preceding code snippet is the default security password printed in the log.

We can use Postman to fire a request with basic authentication. The following screenshot shows how basic authentication details can be sent along with a request:




As you can see, authentication succeeds and we get a proper response back.







We can configure the user ID and password of our choice in `application.properties`, as shown here:

```
security.user.name=user-name
security.user.password=user-password
```

Spring Security also provides options to authenticate with LDAP or JDBC or any other data source with user credentials.

## Integration testing

The integration test we wrote for the service earlier will start failing because of invalid

credentials. We will now update the integration test to supply basic authentication credentials:

```
private TestRestTemplate template = new TestRestTemplate();
HttpHeaders headers = createHeaders("user-name", "user-password");

HttpHeaders createHeaders(String username, String password) {
  return new HttpHeaders() {
    {
      String auth = username + ":" + password;
      byte[] encodedAuth = Base64.getEncoder().encode
      (auth.getBytes(Charset.forName("US-ASCII")));
      String authHeader = "Basic " + new String(encodedAuth);
      set("Authorization", authHeader);
    }
  };
 }

@Test
public void retrieveTodos() throws Exception {
  String expected = "["
  + "{id:1,user:Jack,desc:\"Learn Spring MVC\",done:false}" + ","
  + "{id:2,user:Jack,desc:\"Learn Struts\",done:false}" + "]";
  ResponseEntity<String> response = template.exchange(
  createUrl("/users/Jack/todos"), HttpMethod.GET,
  new HttpEntity<String>(null, headers),
  String.class);
  JSONAssert.assertEquals(expected, response.getBody(), false);
}
```

Some important things to note are as follows:

- `createHeaders("user-name", "user-password")`: This method creates `Base64. getEncoder().encode` basic authentication headers







- `ResponseEntity<String> response = template.exchange(createUrl("/users/Jack/todos"), HttpMethod.GET,new HttpEntity<String>(null, headers), String.class)`: The key change is the use of `HttpEntity` to supply the headers that we created earlier to the REST template

## Unit testing

We would not want to use security for our unit tests. The following code snippet shows how we can disable security for the unit test:

```
@RunWith(SpringRunner.class)
@WebMvcTest(value = TodoController.class, secure = false)
public class TodoControllerTest {
```

The key part is the `secure = false` parameter on the `WebMvcTest` annotation. This will disable Spring Security for the unit test.

# OAuth 2 authentication

OAuth is a protocol that provides flows in order to exchange authorization and authentication information between a range of web-enabled applications and services. It enables third-party applications to get restricted access to user information from a service, for example, Facebook, Twitter, or GitHub.

Before we get into the details, it would be useful to review the terminology typically used with respect to OAuth 2 authentication.

Let's consider an example. Let's say we want to expose the Todo API to third-party applications on the internet.

The following are the important players in a typical OAuth 2 exchange:

- **Resource owner**: This is the user of the third-party application that wants to use our Todo API. It decides how much of the information available with our API can be made available to the third-party application.
- **Resource server**: This hosts the Todo API, the resource we want to secure.
- **Client**: This is the third-party application that wants to consume our API.
- **Authorization server**: This is the server that provides the OAuth service.









# Caching

Caching data from services plays a crucial role in improving the performance and scalability of applications. In this section, we will look at the implementation options that Spring Boot provides.

Spring provides a caching abstraction based on annotations. We will start with using Spring caching annotations. Later, we will introduce *JSR-107* caching annotations and compare them with Spring abstractions.

## Spring-boot-starter-cache

Spring Boot provides a starter project for caching `spring-boot-starter-cache`. Adding this to an application brings in all the dependencies to enable *JSR-107* and Spring caching annotations. The following code snippet shows the dependency details for `spring-boot-starter-cache`. Let's add this to our file `pom.xml`:

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

## Enabling caching

Before we can start using caching, we need to enable caching on the application. The following code snippet shows how we can enable caching:

```
@EnableCaching
@SpringBootApplication
public class Application {
```

`@EnableCaching` would enable caching in a Spring Boot application.

Spring Boot automatically configures a suitable CacheManager framework to serve as a provider for the relevant cache. We will look at the details of how Spring Boot decides the CacheManager a little later.







## Caching data

Now that we have enabled caching, we can add the `@Cacheable` annotation to the methods where we want to cache the data. The following code snippet shows how to enable caching on `retrieveTodos`:

```
@Cacheable("todos")
public List<Todo> retrieveTodos(String user) {
```

In the preceding example, the `todos` for a specific user are cached. On the first call to the method for a specific user, the `todos` will be retrieved from the service. On subsequent calls for the same user, the data will be returned from the cache.

Spring also provides conditional caching. In the following snippet, caching is enabled only if the specified condition is satisfied:

```
@Cacheable(cacheNames="todos", condition="#user.length < 10")
public List<Todo> retrieveTodos(String user) {
```

Spring also provides additional annotations to evict data from the cache and add some custom data to cache. A few important ones are listed as follows:

- `@CachePut`: Used to explicitly add data to the cache
- `@CacheEvict`: Used to remove stale data from the cache
- `@Caching`: Allows multiple nested `@Cacheable`, `@CachePut` , and `@CacheEvict` annotations to be used on the same method

## JSR-107 caching annotations

*JSR-107* aims to standardize caching annotations. Listed here are some of the important *JSR-107* annotations:

- `@CacheResult`: Similar to `@Cacheable`
- `@CacheRemove`: Similar to `@CacheEvict`; `@CacheRemove` supports conditional eviction if an exception occurs
- `@CacheRemoveAll`: Similar to `@CacheEvict(allEntries=true)`; used to remove all entries from the cache

*JSR-107* and Spring's caching annotations are fairly similar in terms of the features they offer. Either of them is a good choice. We lean slightly toward *JSR-107* because it's a standard. However, make sure you are not using both in the same project.







## Auto-detection order

When caching is enabled, Spring Boot auto-configuration starts looking for a caching provider. The following list shows the order in which Spring Boot searches for caching providers. The list is in order of decreasing preference:

- JCache (*JSR-107*) (EhCache 3, Hazelcast, Infinispan, and so on)
- EhCache 2.x
- Hazelcast
- Infinispan
- Couchbase
- Redis
- Caffeine
- Guava
- Simple

## Summary

Spring Boot makes developing Spring-based applications easy. It enables us to create production-ready applications very quickly.

In this chapter, we covered how to add features such as exception handling, caching, and internationalization to our application. We discussed the best practices of documenting REST services using Swagger. We looked at the basics of securing our microservice with Spring Security.

In the next chapter, we will shift our attention toward advanced features in Spring Boot. We will look at how to provide monitoring on top of our REST services, learn how to deploy the microservice to the Cloud, and understand how to become more productive when developing applications with Spring Boot.

# Advanced Spring Boot Features

In the previous chapter, we extended our microservice with exception handling, HATEOAS, caching, and internationalization. In this chapter, let's turn our attention to deploying our services to production. To be able to deploy the services to production, we need to be able to set up and create functionality to configure, deploy, and monitor services.

The following are some of the questions we will answer during this chapter:

- How to externalize application configuration?
- How to use profiles to configure environment-specific values?
- How to deploy our application to the Cloud?
- What is an embedded server? How can you use Tomcat, Jetty, and Undertow?
- What monitoring features does Spring Boot Actuator provide?
- How can you be a more productive developer with Spring Boot?



Pages 230 to 236 are not shown in this preview.



```
    }
    public String getService1Url() {
      return service1Url;
    }
    public void setService1Url(String service1Url) {
      this.service1Url = service1Url;
    }
    public int getService1Timeout() {
      return service1Timeout;
    }
    public void setService1Timeout(int service1Timeout) {
      this.service1Timeout = service1Timeout;
    }
  }
```

A couple of important things to note are as follows:

- `@ConfigurationProperties("application")` is the annotation for an externalized configuration. We can add this annotation to any class to bind to external properties. The value in the double quotes--application--is used as a prefix while binding external configuration to this bean.
- We are defining multiple configurable values in the bean.
- Getters and setters are needed since binding happens through Java beans property descriptors.

The following snippet shows how the values for these properties can be defined in `application.properties`:

```
application.enableSwitchForService1=true
application.service1Url=http://abc-dev.service.com/somethingelse
application.service1Timeout=250
```

A couple of important things to note are as follows:

- `application`: The prefix is defined as part of `@ConfigurationProperties("application")` while defining the configuration bean
- Values are defined by appending the prefix to the name of the property





Page 238 is not part of this book preview.



```
required type 'int' for property 'service1Timeout'; nested exception is
org.springframework.core.convert.ConverterNotFoundException: No converter
found capable of converting from type [java.lang.String] to type [int]

Action:
Update your application's configuration
```

## Profiles

Until now, we looked at how to externalize application configuration to a property file, `application.properties`. What we want to be able to do is have different values for the same property in different environments.

Profiles provide a way to provide different configurations in different environments.

The following snippet shows how to configure an active profile in `application.properties`:

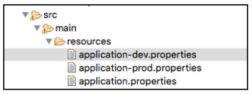
```
spring.profiles.active=dev
```

Once you have an active profile configured, you can define properties specific to that profile in `application-{profile-name}.properties`. For `dev` profile, the name of the properties file would be `application-dev.properties`. The following example shows the configuration in `application-dev.properties`:

```
application.enableSwitchForService1=true
application.service1Url=http://abc-dev.service.com/somethingelse
application.service1Timeout=250
```

The values in `application-dev.properties` will override the default configuration in `application.properties` if the active profile is `dev`.

We can have configurations for multiple environments, as shown here:










## Profiles-based Bean configuration

Profiles can also be used to define different beans or different bean configurations in different environments. All classes marked with `@Component` or `@Configuration` can also be marked with an additional `@Profile` annotation to specify the profile in which the bean or configuration is enabled.

Let's consider an example. An application needs different caches enabled in different environments. In the `dev` environment, it uses a very simple cache. In production, we would want to use a distributed cache. This can be implemented using profiles.

The following bean shows the configuration enabled in a `dev` environment:

```
@Profile("dev")
@Configuration
public class DevSpecificConfiguration {
  @Bean
  public String cache() {
    return "Dev Cache Configuration";
  }
}
```

The following bean shows the configuration enabled in a production environment:

```
@Profile("prod")
@Configuration
public class ProdSpecificConfiguration {
  @Bean
  public String cache() {
    return "Production Cache Configuration - Distributed Cache";
  }
}
```

Based on the active profile configured, the respective configuration is picked up. Note that we are not really configuring a distributed cache in this example. We are returning a simple string to illustrate that profiles can be used to implement these kinds of variations.

## Other options for application configuration values

Until now, the approaches we took to configure application properties was using the key value pairs from either `application.properties` or `application-{profile-name}.properties`.





Pages 241 to 250 are not shown in this preview.

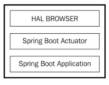


## HAL Browser

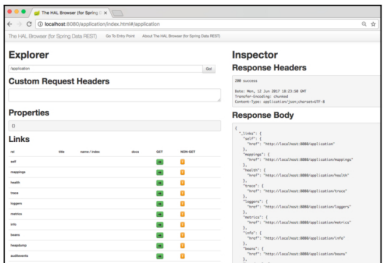
A number of these endpoints expose a lot of data. To be able to visualize the information better, we will add an **HAL Browser** to our application:

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
```

Spring Boot Actuator exposes REST APIs around all the data captured from the Spring Boot application and environment. The HAL Browser enables visual representation around the Spring Boot Actuator API:



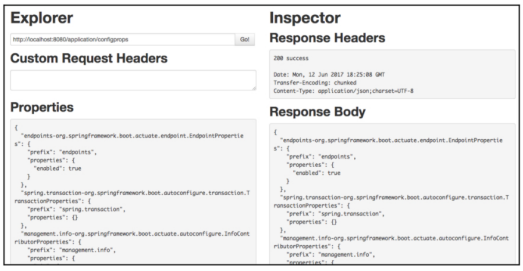

When we launch `http://localhost:8080/application` in the browser, we can see all the URLs exposed by actuator.



Let's browse all the information exposed by actuator as part of different endpoints through the HAL Browser.

## Configuration properties

The `configprops` endpoint provides information about configuration options that can be configured through application properties. It basically is a collated list of all `@ConfigurationProperties`. The following screenshot shows configprops in HAL Browser:










To illustrate a known example, the following section from the service response shows the configuration options available for Spring MVC:

```
"spring.mvc-  org.springframework.boot.autoconfigure.web.WebMvcProperties":
{
   "prefix": "spring.mvc",
   "properties": {
                   "dateFormat": null,
                   "servlet": {
                     "loadOnStartup": -1
                   },
   "staticPathPattern": "/**",
   "dispatchOptionsRequest": true,
   "dispatchTraceRequest": false,
   "locale": null,
   "ignoreDefaultModelOnRedirect": true,
   "logResolvedException": true,
   "async": {
             "requestTimeout": null
           },
   "messageCodesResolverFormat": null,
   "mediaTypes": {},
   "view": {
            "prefix": null,
            "suffix": null
          },
   "localeResolver": "ACCEPT_HEADER",
   "throwExceptionIfNoHandlerFound": false
    }
 }
```

To provide configuration for Spring MVC, we combine the prefix with the path in properties. For example, to configure `loadOnStartup`, we use a property with the name `spring.mvc.servlet.loadOnStartup`.







## Environment details

The **environment** (**env**) endpoint provides information about the operating system, JVM installation, classpath, system environment variable, and the values configured in various application properties files. The following screenshot shows the environment endpoint in the HAL Browser:










Some of the important Spring Data features are listed as follows:

- Easy integration with multiple data stores through various repositories
- The ability to parse and form queries based on repository method names
- Provides the default CRUD functionality
- Basic support for auditing, such as created by user and last changed by user
- Powerful integration with Spring
- Great integration with Spring MVC to expose REST controllers through **Spring Data Rest**

Spring Data is an umbrella project made up of a number of modules. A few of the important Spring Data modules are listed as follows:

- **Spring Data Commons**: Defines the common concepts for all Spring Data modules--repository and query methods
- **Spring Data JPA**: Provides easy integration with JPA repositories
- **Spring Data MongoDB**: Provides easy integration with MongoDB--a document-based data store
- **Spring Data REST**: Provides the functionality to expose Spring Data repositories as REST services with minimal code
- **Spring Data for Apache Cassandra**: Provides easy integration with Cassandra
- **Spring for Apache Hadoop**: Provides easy integration with Hadoop

In this chapter, we will take an in-depth look at the common concepts behind Spring Data, repository, and query methods. In the initial examples, we will use Spring Data JPA to illustrate these concepts. We will also take a look at a sample integration with MongoDB later in the chapter.

## Spring Data Commons

Spring Data Commons provides the basic abstractions behind Spring Data modules. We will use Spring Data JPA as an example to illustrate these abstractions.

Some of the important interfaces in Spring Data Commons are listed as follows:

```
Repository<T, ID extends Serializable>
CrudRepository<T, ID extends Serializable> extends Repository<T, ID>
PagingAndSortingRepository<T, ID extends Serializable> extends
CrudRepository<T, ID>
```







## Repository

Repository is the core interface of Spring Data. It is a **marker interface**.

## The CrudRepository interface

The `CrudRepository` defines the basic `Create`, `Read`, `Update`, and `Delete` methods. The important methods in `CrudRepository` are shown in the following code:

```
public interface CrudRepository<T, ID extends Serializable>
  extends Repository<T, ID> {
  <S extends T> S save(S entity);
  findOne(ID primaryKey);
  Iterable<T> findAll();
  Long count();
  void delete(T entity);
  boolean exists(ID primaryKey);
  // … more functionality omitted.
}
```

## The PagingAndSortingRepository interface

The PagingAndSortingRepository defines methods that provide the functionality to divide the ResultSet into pages as well as sort the results:

```
public interface PagingAndSortingRepository<T, ID extends
  Serializable>
  extends CrudRepository<T, ID> {
    Iterable<T> findAll(Sort sort);
    Page<T> findAll(Pageable pageable);
}
```

We will look at examples of using the `Sort` class and `Page`, `Pageable` interfaces in the section on Spring Data JPA.

## Spring Data JPA

Spring Data JPA implements the core functionality defined in Spring Data Common interfaces.







**JpaRepository** is the JPA-specific repository interface:

```
public interface JpaRepository<T, ID extends Serializable>
extends PagingAndSortingRepository<T, ID>,
QueryByExampleExecutor<T>     {
```

**SimpleJpaRepository** is the default implementation of the CrudRepository interface for JPA:

```
public class SimpleJpaRepository<T, ID extends Serializable>
implements JpaRepository<T, ID>, JpaSpecificationExecutor<T>
```

## Spring Data JPA example

Let's set up a simple project to understand the different concepts related to Spring Data Commons and Spring Data JPA.

The following are the steps involved:

1. Create a new project with `spring-boot-starter-data-jpa` as a dependency.
2. Add entities.
3. Add the `SpringBootApplication` class to run the application.
4. Create repositories.

### New project with Starter Data JPA

We will create a simple Spring Boot Maven project using the following dependencies:

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
   <groupId>com.h2database</groupId>
   <artifactId>h2</artifactId>
   <scope>runtime</scope>
</dependency>
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test</artifactId>
   <scope>test</scope>
</dependency>
```





Pages 270 to 280 are not shown in this preview.



We will start with examples related to finding rows matching specific attribute values. The following example shows different methods in order to search for the `User` by their name:

```
public interface UserRepository
extends PagingAndSortingRepository<User, Long> {
  List<User> findByName(String name);
  List<User> findByName(String name, Sort sort);
  List<User> findByName(String name, Pageable pageable);
  Long countByName(String name);
  Long deleteByName(String name);
  List<User> removeByName(String name);
}
```

Important things to note are as follows:

- `List<User> findByName(String name)`: The pattern is `findBy`, followed by the name of the attribute that you would want to query by. The value of the attribute is passed in as a parameter.
- `List<User> findByName(String name, Sort sort)`: This method allows you to specify a specific sort order.
- `List<User> findByName(String name, Pageable pageable)`: This method allows the use of pagination.
- Instead of find we can also use read, query or get to name the methods. For example, queryByName instead of findByName.
- Similar to find..By we can use count..By to find the count, and delete..By (or remove..By) to delete records.

The following example shows how to search by attributes of a containing element:

```
List<User> findByTodosTitle(String title);
```

The user contains `Todos`. `Todo` has `title` attribute. To create a method to search a user based on the title of the todo, we can create a method by the name `findByTodosTitle` in `UserRepository`.

The following examples show a few more variations that are possible with `findBy`:

```
public interface TodoRepository extends CrudRepository<Todo, Long>
{
  List<Todo> findByTitleAndDescription
  (String title, String description);
  List<Todo> findDistinctTodoByTitleOrDescription
  (String title,String description);
  List<Todo> findByTitleIgnoreCase(String title, String
  description);
```

```
    List<Todo> findByTitleOrderByIdDesc(String lastname);
    List<Todo> findByIsDoneTrue(String lastname);
}
```

Important things to note are as follows:

- `findByTitleAndDescription`: Multiple attributes can be used to query
- `findDistinctTodoByTitleOrDescription`: Find distinct rows that match either title or description
- `findByTitleIgnoreCase`: Illustrates the use of the ignore case
- `findByTitleOrderByIdDesc`: Illustrates an example of specifying a specific sort order

The following example shows how to find a specific subset of records using find:

```
public interface UserRepository
extends PagingAndSortingRepository<User, Long> {
  User findFirstByName(String name);
  User findTopByName(String name);
  List<User> findTop3ByName(String name);
  List<User> findFirst3ByName(String name);
}
```

Important things to note are as follows:

- `findFirstByName, findTopByName`: Queries for the first user
- `findTop3ByName, findFirst3ByName`: Finds the top three users

## Queries

Spring Data JPA also provides options to write custom queries. The following snippet shows a simple example:

```
@Query("select u from User u where u.name = ?1")
List<User> findUsersByNameUsingQuery(String name);
```

Important things to note are as follows:

- `@Query`: The annotation to define queries for repository methods
- `select u from User u where u.name = ?1`: `Query to be executed. ?1` represents the first parameter
- `findUsersByNameUsingQuery`: When this method is called, the query specified is executed with the name as the parameter





Pages 283 to 284 are not shown in this preview.



Important things to note are as follows:

- `@RepositoryRestResource`: The annotation used to expose a repository using REST
- `collectionResourceRel = "users"`: The `collectionResourceRel` value to be used in the generated links
- `path = "users"`: The path under which the resource has to be exposed

When we launch `SpringDataJpaFirstExampleApplication` as a Java application, the following can be seen in the log:

```
s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s):
8080 (http)
o.s.b.w.servlet.ServletRegistrationBean : Mapping servlet:
'dispatcherServlet' to [/]
o.s.b.w.servlet.FilterRegistrationBean : Mapping filter:
'characterEncodingFilter' to: [/*]
s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto ****
o.s.d.r.w.RepositoryRestHandlerMapping : Mapped "{[/{repository}],
methods=[OPTIONS]
o.s.d.r.w.RepositoryRestHandlerMapping : Mapped "{[/{repository}],
methods=[HEAD]
o.s.d.r.w.RepositoryRestHandlerMapping : Mapped "{[/{repository}],
methods=[GET]
o.s.d.r.w.RepositoryRestHandlerMapping : Mapped "{[/{repository}],
methods=[POST]
o.s.d.r.w.RepositoryRestHandlerMapping : Mapped "{[/{repository}/{id}],
methods=[OPTIONS]
o.s.d.r.w.RepositoryRestHandlerMapping : Mapped
"{[/{repository}/{id}/{property}]
o.s.d.r.w.RepositoryRestHandlerMapping : Mapped "{[/{repository}/search],
methods=[GET]
```

The preceding log shows that the Spring MVC DispatcherServlet is launched and ready to serve different request methods and URIs.





Pages 286 to 299 are not shown in this preview.



Important things to note are as follows:

- `@Value("${number.service.url}") private String numberServiceUrl`: We would want the number service URL to be configurable in application properties.
- `@RequestMapping("/add") public Long add()`: Exposes a service at the URI `/add`. The `add` method calls the number service using `RestTemplate` and has the logic to sum the numbers returned in the response.

Let's configure `application.properties`, as shown in the following snippet:

```
spring.application.name=service-consumer
server.port=8100
```

```
number.service.url=http://localhost:8080/random
```

Important things to note are as follows:

- `spring.application.name=service-consumer`: Configures a name for the Spring Boot application
- `server.port=8100`: Uses `8100` as the port for service consumer
- `number.service.url=http://localhost:8080/random`: Configures the number service URL for use in the add service

When the service is called at the URL `http://localhost:8100/add`, the following response is returned:

```
2890
```

The following is an extract from the log of Microservice A:

```
c.m.s.c.c.RandomNumberController : Returning [752,
  119, 493, 871, 445]
```

The log shows that `random service` from Microservice A returned `5` numbers. The `add` service in the service consumer added them up and returned a result `2890`.

We now have our example microservices ready. In the next steps, we will add Cloud-Native features to these microservices.

## Ports

In this chapter, we will create six different microservices applications and components. To keep things simple, we will use specific ports for specific applications.









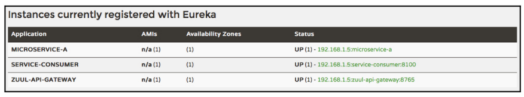
- `public boolean shouldFilter()`: If the filter should only be executed in certain conditions, the logic can be implemented here. If you would want the filter to always be executed, return `true`.
- `public Object run()`: The method to implement the logic for the filter. In our example, we are logging the request method and the URL of the request.

When we start up the Zuul server by launching `ZuulApiGatewayServerApplication` as a Java application, you will see the following log in `Eureka Name Server`:

```
Registered instance ZUUL-API-GATEWAY/192.168.1.5:zuul-api-
  gateway:8765 with status UP (replication=false)
```

This shows that `Zuul API gateway` is up and running. `Zuul API gateway` is also registered with `Eureka Server`. This allows microservice consumers to talk to the Name server to get details about `Zuul API gateway`.

The following figure shows the Eureka dashboard at `http://localhost:8761`. You can see that instances of `Microservice A`, `service consumer`, and `Zuul API Gateway` are now registered with `Eureka Server`:




The following is an extract from the `Zuul API gateway` log:

```
Mapped URL path [/microservice-a/**] onto handler of type [
class org.springframework.cloud.netflix.zuul.web.ZuulController]
Mapped URL path [/service-consumer/**] onto handler of type [
class org.springframework.cloud.netflix.zuul.web.ZuulController]
```

By default, all services in Microservice A and the service consumer microservice are enabled for reverse proxying by Zuul.







### Invoking microservices through Zuul

Let's invoke `random service` through the service proxy now. The direct URL to a random microservice is `http://localhost:8080/random`. This is exposed by Microservice A, whose application name is `microservice-a`.

The URL structure to call a service through `Zuul API Gateway` is `http://localhost:{port}/{microservice-application-name}/{service-uri}`. So, the `Zuul API Gateway` URL for `random service` is `http://localhost:8765/microservice-a/random`. When you invoke `random service` through API Gateway, you get a response shown here. The response is similar to what you would typically get when directly calling the random service:

```
[73,671,339,354,211]
```

The following is an extract from the `Zuul Api Gateway` log. You can see that the `SimpleLoggingFilter` that we created in `Zuul API Gateway` is executed for the request:

```
c.m.s.z.filters.pre.SimpleLoggingFilter : Request Method : GET
URL: http://localhost:8765/microservice-a/random
```

The `add` service is exposed by service consumer, whose application name is service-consumer and the service URI is `/add`. So, the URL to execute the `add` service through the API Gateway is `http://localhost:8765/service-consumer/add`. The response from the service is shown here. The response is similar to what you would typically get when directly calling the `add` service:

```
2488
```

The following is an extract from the `Zuul API Gateway` log. You can see that the initial `add` service call is going through the API Gateway:

```
2017-03-28 14:05:17.514 INFO 83147 --- [nio-8765-exec-1]
c.m.s.z.filters.pre.SimpleLoggingFilter : Request Method : GET
URL: http://localhost:8765/service-consumer/add
```

The `add` service calls `random service` on `Microservice A`. While the initial call to add service goes through the API Gateway, the call from add service (service consumer microservice) to `random service` (Microservice A) is not routed through API Gateway. In an ideal world, we would want all the communication to take place through API Gateway.





Pages 333 to 342 are not shown in this preview.



`NumberAdderController` exposes a service with request mapping `/add`. This uses `RandomServiceProxy` to fetch random numbers. What if this service fails? Hystrix provides a fallback. The following snippet shows how we can add a fallback method to a request mapping. All we need to do is add the `@HystrixCommand` annotation to the `fallbackMethod` attribute, defining the name of the fallback method--in this example, `getDefaultResponse`:

```
@HystrixCommand(fallbackMethod = "getDefaultResponse")
@RequestMapping("/add")
public Long add() {
  //Logic of add() method
}
```

Next, we define the `getDefaultResponse()` method with the same return type as the `add()` method. It returns a default hardcoded value:

```
public Long getDefaultResponse() {
  return 10000L;
 }
```

Let's bring down Microservice A and invoke `http://localhost:8100/add`. You will get the following response:

```
10000
```

When `Microservice A` fails, the service consumer microservice handles it gracefully and offers reduced functionality.

## Summary

Spring Cloud makes it easy to add Cloud-Native features to your microservices. In this chapter, we looked at some of the important patterns in developing Cloud-Native applications and implemented them using various Spring Cloud projects.

It is important to remember that the field of developing Cloud-Native applications is still in its inception phase--in its first few years. It would need more time to mature. Expect some evolution in patterns and frameworks in the years to come.

In the next chapter, we will shift our attention to Spring Data Flow. Typical use cases on the cloud include real-time data analytics and data pipelines. These use cases involve the flow of data between multiple microservices. Spring Data Flow provides patterns and best practices for distributed streaming and data pipelines.





# 10
# Spring Cloud Data Flow

Spring Data Flow brings the microservices architecture into typical data flow and event flow scenarios. We will discuss more about these scenarios later in this chapter. Building on top of other Spring Projects, such as Spring Cloud Stream, Spring Integration, and Spring Boot, Spring Data Flow makes it easy to define and scale use cases involving data and event flows using message-based integration.

In this chapter, we will discuss the following topics:

- Why do we need asynchronous communication?
- What is Spring Cloud Stream? How does it build on top of Spring Integration?
- Why do we need Spring Data Flow?
- What are the important concepts in Spring Data Flow you would need to understand?
- What are the use cases where Spring Data Flow is useful?

We will also implement a simple event flow scenario with three microservices acting as the source (application generating the events), processor, and sink (application consuming events). We will implement the microservices using Spring Cloud Stream and establish connections between them over the message broker using Spring Cloud Data Flow.

## Message-based asynchronous communication

There are two options when integrating applications:

- **Synchronous**: Service consumer invokes the service provider and waits for a response.



Pages 345 to 356 are not shown in this preview.

Update the `SpringBootApplication` file with the following code:

```
@EnableBinding(Processor.class)@SpringBootApplication
public class StockIntelligenceProcessorApplication {
  private static Logger logger =
  LoggerFactory.getLogger
  (StockIntelligenceProcessorApplication.class);
  private static Map < StockTicker, Integer > holdings =
    getHoldingsFromDatabase();
    private static Map < StockTicker,
    Integer > getHoldingsFromDatabase() {
      final Map < StockTicker,
      Integer > holdings = new HashMap < >();
      holdings.put(StockTicker.FACEBOOK, 10);
      holdings.put(StockTicker.GOOGLE, 0);
      holdings.put(StockTicker.IBM, 15);
      holdings.put(StockTicker.MICROSOFT, 30);
      holdings.put(StockTicker.TWITTER, 50);
      return holdings;
    }
    @Transformer(inputChannel = Processor.INPUT,
    outputChannel = Processor.OUTPUT)
    public Object addOurInventory(StockPriceChangeEvent event) {
      logger.info("started processing event " + event);
      Integer holding =  holdings.get(
        StockTicker.valueOf(event.getStockTicker()));
      StockPriceChangeEventWithHoldings eventWithHoldings =
        new StockPriceChangeEventWithHoldings(event, holding);
      logger.info("ended processing eventWithHoldings "
        + eventWithHoldings);
      return eventWithHoldings;
    }
    public static void main(String[] args) {
      SpringApplication.run(
        StockIntelligenceProcessorApplication.class,args);
    }
}
```

A few important things to note are as follows:

- `@EnableBinding(Processor.class)`: The `EnableBinding` annotation enables binding a class with the respective channel it needs--an input and/or an output. The `Processor` class is used to register a Cloud Stream with one input channel and one output channel.





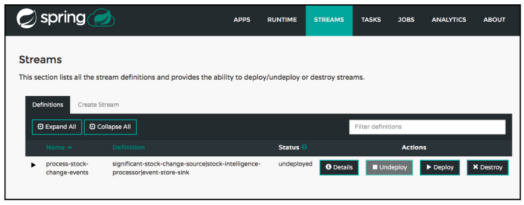
Pages 358 to 369 are not shown in this preview.



You should see the following output if the stream is successfully created:

```
Created new stream 'process-stock-change-events'
```

You can also see the registered stream on the **Streams** tab of Spring Cloud Data Flow dashboard at `http://localhost:9393/dashboard`, as shown in the following screenshot:




## Deploying the stream

To deploy the stream, we can execute the following command on the Data Flow Shell:

```
stream deploy --name process-stock-change-events
```

You will see the message shown here when the request is sent for the creation of stream:

```
Deployment request has been sent for stream 'process-stock-change-events'
```

The following extract shows an extract from the Local Data Flow Server log:

```
o.s.c.d.spi.local.LocalAppDeployer : deploying app process-stock-change-
events.event-store-sink instance 0

Logs will be in /var/folders/y_/x4jdvdkx7w94q5qsh745gzz00000gn/T/spring-
cloud-dataflow-3084432375250471462/process-stock-change-
events-1492100265496/process-stock-change-events.event-store-sink

o.s.c.d.spi.local.LocalAppDeployer : deploying app process-stock-change-
events.stock-intelligence-processor instance 0
```





Pages 371 to 465 are not shown in this preview.