

Spring 5.0

Projects

Build seven web development projects with Spring MVC, Angular 6, JHipster, WebFlux, and Spring Boot 2



Nilang Patel

Packt>

www.packt.com

Spring 5.0 Projects

Build seven web development projects with Spring MVC, Angular 6, JHipster, WebFlux, and Spring Boot 2

Nilang Patel



BIRMINGHAM - MUMBAI

Spring 5.0 Projects

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Aaron Lazar

Acquisition Editor: Denim Pinto

Content Development Editor: Anugraha Arunagiri

Technical Editor: Abin Sebastian

Copy Editor: Safis Editing

Project Coordinator: Ulhas Kambali

Proofreader: Safis Editing

Indexer: Mariammal Chettiyar

Graphics: Tom Scaria

Production Coordinator: Aparna Bhagat

First published: February 2019

Production reference: 1280219

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78839-041-5

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.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.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.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.

Contributors

About the author

Nilang Patel has over 15 years of core IT experience in leading projects, software design and development, and supporting enterprise applications using enterprise Java technologies. He is highly skilled in core Java/J2EE-based applications, and has experience in the healthcare, human resource, taxation, intranet applications, energy, and risk management domains. He contributes to different communities through various forums and his personal blog. He is also the author of *Java 9 Dependency Injection*, and acquired Liferay 6.1 Developer Certification in 2013, Brainbench Java 6 certification in 2012, and became a Sun Certified Programmer for the Java 2 Platform 1.5 (SCJP) in 2007.

About the reviewer

Krunal Patel has been working at Liferay Portal for over six years and has over ten years' experience in enterprise application development using Java and Java EE technologies. He has worked in various domains, including healthcare, hospitality, and enterprise intranet. He was awarded an ITIL Foundation certificate in IT Service Management in 2015, Liferay 6.1 Developer Certification in 2013, and was awarded a MongoDB for Java Developers certificate in 2013. He is the co author of *Java 9 Dependency Injection* book and also reviewed *Mastering Apache Solr 7.x* by Packt Publishing.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface	1
Chapter 1: Creating an Application to List World Countries with their GDP	6
Technical requirements	7
Introduction to the application	7
Understanding the database structure	7
Understanding the World Bank API	8
Designing the wireframes of application screens	10
Country listing page	10
Country detail page	11
Country edit page	12
Add a new city and language	13
Creating an empty application	13
Defining the model classes	15
Using Hibernate Validator to add validations	18
Defining the data access layer – Spring JDBC Template	19
Defining the JDBC connection properties	21
Setting up the test environment	23
Defining the RowMapper	25
Designing the CountryDAO	27
Designing the CityDAO	32
Designing the CountryLanguageDAO	35
Designing the client for World Bank API	37
Defining the API controllers	38
Enabling Web MVC using @EnableWebMvc	38
Configuration to deploy to Tomcat without web.xml	39
Defining the RESTful API controller for country resource	41
Defining the RESTful API controller for city resource	43
Defining the RESTful API controller for country language resource	44
Deploying to Tomcat	44
Defining the view controller	47
Defining the view templates	49
Configuring a Thymeleaf template engine	50
Managing static resources	51
Creating the base template	52
Logging configuration	53
Running the application	54
Summary	58

Chapter 2: Building a Reactive Web Application	59
Technical requirements	60
Reactive system	60
Reactive Programming	61
Basics of Reactive Programming	63
Backpressure	65
Benefits of Reactive Programming	65
Reactive Programming techniques	66
Reactive Programming in Java	67
Reactive Streams	67
Reactive Streams specifications	68
Publisher rules	70
Subscriber rules	71
Subscription rules	72
Processor rules	72
Reactive Streams TCK	72
RxJava	73
Anatomy of RxJava	74
Observer event calls	77
Observable for iterators	79
Custom Observers	80
Observable types	82
Cold Observable	82
Hot Observable	83
Other ways to get Observable	86
Operators	87
Project Reactor	87
Reactor features	88
Handling data stream with high volume	88
Push-pull mechanism	88
Handling concurrency independently	88
Operators	89
Reactor sub-projects	89
Reactor types	89
Reactor in action	90
Types of subscribers	92
Custom subscribers	93
Reactor lifecycle methods	95
Ratpack	97
Akka stream	97
Vert.x	97
Reactive support in Spring Framework	97
Spring WebFlux	98
Spring MVC versus Spring WebFlux	98
Reactive span across Spring modules	100
Spring WebFlux application	101
MongoDB installation	103
MongoDB data structure	103
Creating a Spring Data repository	104

WebFlux programming models	105
Annotated controller	106
Functional endpoint	109
Artifacts required in functional-style Reactive Programming	109
Prerequisite for a functional approach in Spring WebFlux	110
Defining routers and handlers	111
Combining handler and router	113
Composite routers	114
WebSocket support	116
Summary	118
Chapter 3: Blogpress - A Simple Blog Management System	119
Technical requirements	120
Application overview	120
Project skeleton with Spring Boot	121
Configuring IDE Spring Tool Suite	122
Spring Model-View-Controller web flow	124
Presentation layer with Thymeleaf	125
How Thymeleaf works	126
Dialects, processors, and expression objects	126
Why Thymeleaf is a natural template	128
Making the application secure with Spring Security	131
Excluding auto-configuration	133
Substituting auto-configuration	133
Storing data with Elasticsearch	139
Artifacts	140
Documents	140
Indexes	140
Clusters and nodes	141
Shards and replicas	141
Interacting with Elasticsearch	142
Installation	142
Elasticsearch RESTful API	143
Creating an index – students	143
Creating a document type – student	144
Adding a document (student data)	146
Reading a document (student data)	146
Updating a document (student data)	147
Deleting a document (student data)	148
Searching a query	148
Creating index and document types for Blogpress	149
Elasticsearch integration with Spring Data	150
Spring Data Elasticsearch model class	152
Connecting Elasticsearch with Spring Data	154
CRUD operations in Elasticsearch with Spring Data	154
Adding blog data	155
Reading blog data	156
Searching blog data	156

Adding comment data with Elasticsearch aggregation	157
Reading comment data with Elasticsearch aggregation	162
Updating and deleting comment data with Elasticsearch	167
Displaying data with RESTful web services in Spring	168
Building a UI with the Mustache template	169
Summary	175
Chapter 4: Building a Central Authentication Server	177
Technical requirements	178
LDAP	178
What is LDAP?	179
Configuring Apache DS as an LDAP server	181
Example DIT structures	183
Apache DS partitions	185
The LDAP structure	187
Spring Security integration with LDAP	189
Creating a web application with Spring Boot	190
Managing LDAP users with Spring Data	195
Spring Data models	195
The Spring Data repository for LDAP	197
Performing CRUD operations with LdapTemplate	200
Initializing LdapTemplate	201
Using LdapTemplate to perform CRUD operations	201
LDAP authorization with Spring Security	206
Creating roles in the LDAP server	207
Importing role information to perform authorization	208
OAuth	210
OAuth roles	211
Grant types	213
Authorization code	213
Implicit	215
Resource Owner Password Credentials	216
Client Credentials	217
Which grant type should be used?	218
Spring Security integration with OAuth	218
Application registration	219
Changes in the Spring Boot application	220
The default OAuth configuration	221
OAuth with a custom login page	223
Dual authentication with OAuth and LDAP	225
OAuth authorization with a custom authorization server	229
Authorization server configuration	230
Resource server configuration	233
Method-level resource permissions	239
Summary	241
Chapter 5: An Application to View Countries and their GDP using JHipster	243

Technical requirements	244
Introducing JHipster	244
Installing JHipster	246
Creating an application	247
Project structure	250
Entity creation	253
Adding an entity with the CLI	254
Modeling the entity	258
Modeling with UML	259
Modeling with JHipster Domain Language studio	261
Generating an entity using a model	265
Showing the national gross domestic product	266
Application and entity creation	266
Handling enumeration data with a database in JHipster	269
Filter provision in service, persistence, and the REST controller layer	274
The persistence layer	274
The service layer	274
The REST controller layer	276
Adding a filter option to existing entities	278
Developing custom screens	279
The search country screen	279
Creating an Angular service	280
Creating the Angular router	281
Angular modules	282
Creating an Angular component to show the country list	283
Angular template to show the country list	285
Showing the GDP screen	287
An Angular component to show country GDP	288
Angular template to show country GDP	290
Hooking the GDP module into AppModule	291
Updating navigation	292
Other JHipster features	293
IDE support	293
Setting screens out of the box	294
Home and login screens	294
Account management	295
Administration	295
User management	296
Metrics	296
Health	296
Configuration	296
Audit	297
Logs	297
API	297
Maintaining code quality	297
Microservice support	298
Docker support	298
Profile management	299
Live reload	299

Testing support	300
Upgrading JHipster	300
Continuous integration support	301
Community support and documentation	302
The JHipster Marketplace	302
Summary	303
Chapter 6: Creating an Online Bookstore	304
Technical requirements	305
Microservices introduction	305
Microservice architecture	307
Microservice principles	310
High cohesion with a single responsibility	311
Service autonomy	311
Loose coupling	311
Hide implementation through encapsulation	312
Domain-driven design	312
Microservice characteristics	312
Microservices with Spring Cloud	313
Configuration management	313
Service discovery	313
Circuit breakers	314
Routing	314
Spring Cloud Security	314
Distributed tracing service	315
Spring Cloud Stream	315
Developing an online bookstore application	315
Application architecture	315
Database design	316
Single monolithic database for all microservices	317
Separate service to handle database interaction	318
Each microservice has its own database	319
User schema	320
Order schema	320
Catalog schema	321
Inventory schema	321
Creating microservices with Spring Boot	322
Adding microservice-specific capabilities	323
Develop a service discovery server	324
Designing an API gateway	329
Setting up Zuul as an API gateway	329
Designing the UI	330
Monolithic front	331
Micro front	332
Composite front	333
Other Spring Cloud and Netflix OSS components	335
Dynamic configuration in Spring Cloud	335
Step 1 – creating a Spring Boot service for the configuration server	336
Step 2 – configuring Spring Cloud Config with a Git repository	336

Step 3 – making each microservice Spring Cloud Config-aware using the Spring Cloud Config Client component	338
Making RESTful calls across microservices with Feign	340
Load balancing with Ribbon	342
Configuring Ribbon without Eureka	342
Configuring Ribbon with Eureka	345
Load balancing using RestTemplate	346
Configuring the API gateway	347
Securing an application	350
Summary	356
Chapter 7: Task Management System Using Spring and Kotlin	358
Technical requirements	359
Introducing Kotlin	359
Interoperability	359
Concise yet powerful	360
Safety feature	361
IDE Support	361
Kotlin features	362
The concept of a function	362
Function as an expression	363
Default function arguments	363
Extension functions	364
Lambda expression or function literal	366
Passing lambda to another function	367
Returning a function from another function	369
Null safety	372
Data classes	374
Interoperability	376
Calling the Kotlin code from Java	377
Calling Java code from Kotlin	377
Smart casts	378
Operator overloading	379
Kotlin versus Java	380
Spring supports for Kotlin	381
Developing an application – Task Management System	381
Creating a Spring Boot project with Kotlin	382
DB design	384
Entity classes	385
Users	385
Role	386
Task	387
Comments	388
Spring Security	388
Query approach	390
UserDetailsService approach	391
Defining the Spring MVC controller	395
Showing the control page	397
Showing the login page	397

Table of Contents

Showing the add new task page	398
Showing the edit task page	398
Adding a new task	400
Updating a task	400
Adding a task comment	401
Getting all users	402
Showing a task list	403
Viewing a task	405
Deleting a task	407
REST call in Kotlin	407
Validation	409
User registration	410
Summary	411
Other Books You May Enjoy	412
Index	415

Preface

Spring makes it simple to create RESTful applications, interact with social service, communicate with modern databases, secure your system, and make your code modular and easy to test. This book will show you how to build various projects in Spring 5.0, using its various features, as well as third-party tools.

Who this book is for

This book is for competent Spring developers who wish to understand how to develop complex yet flexible applications with Spring. You must have a good knowledge of Java programming and be familiar with the basics of Spring.

What this book covers

Chapter 1, *Creating an Application to List World Countries with their GDP*, is about kick-starting your Spring-based web application development. We will focus on creating a web application using Spring MVC, Spring Data, and the World Bank API for some statistics on different countries, and a MySQL database. The core data for the application will be from the sample world database that comes with MySQL. The UI for this application will be powered by Angular.js. We will follow the WAR model for application deployment and will deploy on the latest version of Tomcat.

Chapter 2, *Building a Reactive Web Application*, is about building a RESTful web services application purely using Spring's new WebFlux framework. Spring WebFlux is a new framework that helps create reactive applications in a functional way.

Chapter 3, *Blogpress – A Simple Blog Management System*, is about creating a simple Spring Boot-based blog management system that uses Elasticsearch as the data store. We will also implement user roles management, authentication, and authorization using Spring Security.

Chapter 4, *Building a Central Authentication Server*, is about building an application that will act as an authentication and authorization server. We will make use of the OAuth2 protocol and LDAP to build a central application that supports authentication and authorization.

Chapter 5, *Application to View Countries and Their GDP Using JHipster*, revisits the application we developed in Chapter 1, *Creating an Application to List World Countries with their GDP*, and we'll develop the same application using JHipster. JHipster helps with the development of Spring Boot and Angular.js production-ready applications, and we will explore the platform and learn about its features and functionality.

Chapter 6, *Creating an Online Bookstore*, is about creating an online store that sells books by developing a web application in a layered fashion.

Chapter 7, *Task Management System Using Spring and Kotlin*, looks at building a task management system using Spring Framework and Kotlin.

To get the most out of this book

A good understanding of Java, Git, and Spring Framework is necessary before reading this book. A deep knowledge of OOP is desired, although some key concepts are reviewed in the first two chapters.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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/Spring 5.0 Projects>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781788390415_ColorImages.pdf.

Code in Action

Visit the following link to check out videos of the code being run: <http://bit.ly/2ED57Ss>

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The preceding line has to be added between the `<Host></Host>` tags."

A block of code is set as follows:

```
<depedency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</depedency>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<depedency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</depedency>
```

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

```
$ mvn package
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Download STS, unzip it in your local folder, and open the .exe file to start the STS. Once started, create a new **Spring Starter Project** of the **Spring Boot** type with the following attributes."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Creating an Application to List World Countries with their GDP

Spring is an ecosystem that facilitates the development of JVM-based enterprise applications. And this is achieved using various modules provided by Spring. One of them, called Spring-core, is the heart of the framework in the Spring ecosystem, which provides support for dependency injection, web application, data access, transaction management, testing, and others.

In this chapter, we will start from scratch and use Spring Framework to develop a simple application. Familiarity with Spring Framework is not required and we will see to it that by the end of the chapter you should be confident enough to use Spring Framework.

The following are the topics covered in this chapter:

- Introduction to the application
- Understanding the database structure
- Understanding the World Bank API
- Designing the wireframes
- Creating an empty application
- Defining the model classes
- Defining the data access layer
- Defining the API controllers
- Deploying to Tomcat
- Defining the view controllers
- Defining the views

Technical requirements

All the code used in this chapter can be downloaded from the following GitHub link:

<https://github.com/PacktPublishing/Spring-5.0-Projects/tree/master/chapter01>.

The code can be executed on any operating system, although it has only been tested on Windows.

Introduction to the application

We will develop an application to show the GDP information of various countries. We will make use of the sample World DB (<https://dev.mysql.com/doc/world-setup/en/world-setup-installation.html>) available with MySQL to list the countries and get a detailed view to display the country information and its GDP information obtained from the World Bank API (<https://datahelpdesk.worldbank.org/knowledgebase/articles/898599-api-indicator-queries>).

The listing will make use of the countries data available in the World DB. In the detail view, we will make use of data available in the World DB to list cities and languages, and make use of the World Bank API to get additional information and the GDP information about the country.

We will also support editing basic details of the country entry, adding and deleting cities from the country entry, and adding and deleting languages from the country entry. We will use the following tools and technologies in this application:

- Spring MVC framework for implementing the MVC pattern
- The interaction with the MySQL DB will be done using the Spring JDBC template
- The interaction with the World Bank API will be done using RestTemplate
- The views will be created using a templating framework called Thymeleaf
- The frontend will be driven by jQuery and Bootstrap

Understanding the database structure

If you don't have MySQL installed, head over to the MySQL link (<https://dev.mysql.com/downloads/installer>) to install it and populate it with the world database, if it is not already available. The appendix will also guide you on how to run the queries using MySQL Workbench and MySQL command-line tool.

The world database schema is depicted in the following diagram:



The database schema is simple, containing three tables as follows:

- **city**: List of cities mapped to the three character country coded in the country table.
- **country**: List of countries where the primary key is the three character country code. There is a column that has the ISO country code.
- **countrylanguage**: List of languages mapped to the country with one of the languages of the country marked as official.

Understanding the World Bank API

There are a lot of APIs exposed by the World Bank (<http://www.worldbank.org/>) and the API documentation can be found here (<https://datahelpdesk.worldbank.org/knowledgebase/articles/889386-developer-information-overview>). Out of the available APIs, we will use the Indicator APIs (<https://datahelpdesk.worldbank.org/knowledgebase/articles/898599-api-indicator-queries>), which represent information such as total population, GDP, GNI, energy use, and much more.

Using the Indicator API, we will fetch the GDP information for the countries available in the database for the last 10 years. Let's look at the API's REST URL and the data returned by the API, as follows:

```
GET
http://api.worldbank.org/countries/BR/indicators/NY.GDP.MKTP.CD?format=json
&date=2007:2017
```

The BR is a country code (*Brazil*) in this URL. The NY.GDP.MKTP.CD is the flag used by the Word Bank API internally to call Indicator API. The request parameter, *date*, indicates the duration of which the GDP information is required.

The excerpt from the response you will get for the preceding API is as follows:

```
[
  {
    "page": 1,
    "pages": 1,
    "per_page": "50",
    "total": 11
  },
  [
    ....// Other country data
    {
      "indicator": {
        "id": "NY.GDP.MKTP.CD",
        "value": "GDP (current US$) "
      },
      "country": {
        "id": "BR",
        "value": "Brazil"
      },
      "value": "1796186586414.45",
      "decimal": "0",
      "date": "2016"
    }
  ]
]
```

The preceding response shows the GDP indicator in US\$ for Brazil for the year 2016.

A wireframe is the basic skeleton of an application or website. It gives an idea about how the final application looks. It basically helps to decide navigation flows, understand functionality, design the user interface, and helps in setting the expectation before the application even exists. This process greatly helps developers, designers, product owners, and clients to work in a synchronous manner to avoid any gap in between. We will follow the same model and we will design various wireframes of the application as follows.

We will make it simple. The home page shows the country list with pagination, and allow searching by country name and filtering by continent/region. The following would be the home page of our application:

WORLD IN NUMBERS

Countries

Search by Name

Search

Continent

Region

All

All

Code

Name

Continent

Region

Area

<<

1

2

3

4

>>

Country detail page

This screen will show details of the country such as cities, languages, and the GDP information obtained from the World Bank API. The GDP data will be visible in a graphical view. The page looks as follows:

WORLD IN NUMBERS

Countries / Brazil

Brazil (BRA/BR)

Edit

FLAG

Capital :

Continent :

Region :

Head of State :

Government :

Local Name :

Independence :

Surface Area :

Population :

Life Expentancy :

Cities

+ New

X

X

X

X

Languages

+

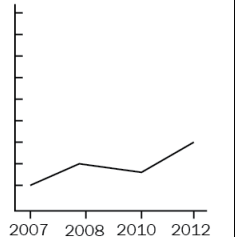
X

X

X

X

GDP



[11]

Country edit page

In country listing page, there will be one button called **Edit**. On clicking it, the system will show the country in edit mode, enabling the update of the basic details of the country. The following is the view structure for editing the basic detail of a country:

WORLD IN NUMBERS

Edit Brazil

Name

Local Name

Capital

Continent

Region

Head of State

Government

Independence

Surface Area

Population

Life Expectancy

× Cancel

+ Save

Add a new city and language

In the country detail page, two modal views, one for adding a new city and another for adding a new language, are available by clicking on the **New** button. The following is the view for the two modal dialogs used to add a new country and language. They will be opened individually:

The image shows two side-by-side modal dialogs. The left dialog is titled 'New City' and contains three text input fields labeled 'Name', 'District', and 'Population'. The right dialog is titled 'New Language' and contains a text input field labeled 'Language', a checkbox labeled 'Is Official', and a text input field labeled 'Percentage'. Both dialogs have a 'Close' button with an 'X' icon and a 'Save' button with a '+' icon at the bottom.

Creating an empty application

We will use Maven to generate an empty application with the structure required for Java-based web applications. If you do not have Maven installed, please follow the instructions here (<https://maven.apache.org/install.html>) to install Maven. Once installed, run the following command to create an empty application:

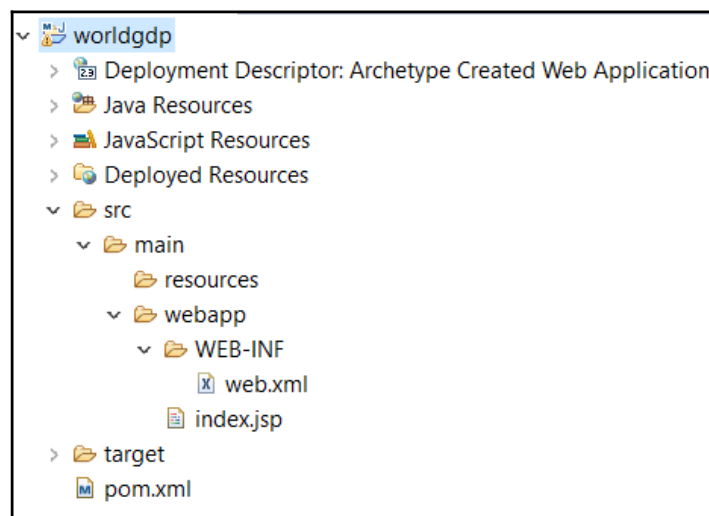
```
mvn archetype:generate -DgroupId=com.nilangpatel.worldgdp -DartifactId=worldgdp -Dversion=0.0.1-SNAPSHOT -DarchetypeArtifactId=maven-archetype-webapp
```

Running the preceding command will show the command-line argument values for confirmation as shown in the following screenshot:

```
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/4/maven-p
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/4/maven-pa
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/archetypes/maven-arche
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/archetypes/maven-archet
kB at 11 kB/s)
[INFO] Using property: groupId = com.nilangpatel.worldgdp
[INFO] Using property: artifactId = worldgdp
[INFO] Using property: version = 0.0.1-SNAPSHOT
[INFO] Using property: package = com.nilangpatel.worldgdp
Confirm properties configuration:
groupId: com.nilangpatel.worldgdp
artifactId: worldgdp
version: 0.0.1-SNAPSHOT
package: com.nilangpatel.worldgdp
Y: :
```

You would have to type in `Y` in the Command Prompt shown in the previous screenshot to complete the empty project creation. Now you can import this project into an IDE of your choice and continue with the development activity. For the sake of simplicity, we will use Eclipse, as it is among the most popular IDEs used by the Java community today.

On successful creation of the application, you will see the folder structure, as shown in the following screenshot:



You will see `index.jsp` added by default while creating the default project structure. You must delete it as, in this application, we will use Thymeleaf—another template engine to develop the landing page.

Defining the model classes

Now, let's create Java classes to model the data in the database and also the data coming from the World Bank API. Our approach is simple. We will have one Java class for each table in our database and the columns of the database will become the properties of the Java class.

In the generated application, the `java` folder is missing under the `main` directory. We will manually create the `java` folder and package the `com.nilangpatel.worldgdp`, which will be the root package for the application. Let's go ahead and implement the approach we decided on. But before that, let's see an interesting project called **Project Lombok**.

Project Lombok provides annotations for generating your getters, setters, default, and overloaded constructors, and other boilerplate code. More details on how to integrate with your IDE can be found on their project website (<https://projectlombok.org/>).

We need to update our `pom.xml` to include a dependency on Project Lombok. The following are the parts of `pom.xml` you need to copy and add to relevant locations in the XML:

```
<properties>
  <java.version>1.8</java.version>
  <lombok.version>1.16.18</lombok.version>
</properties>
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
  <version>${lombok.version}</version>
</dependency>
```

All the model classes that we are going to create next belong to the `com.nilangpatel.worldgdp.model` package. The model class to represent Country data is given in the following code:

```
@Data
@Setter
@Getter
public class Country {
    private String code;
    private String name;
    private String continent;
    private String region;
    private Double surfaceArea;
    private Short indepYear;
```

```
        private Long population;
        private Double lifeExpectancy;
        private Double gnp;
        private String localName;
        private String governmentForm;
        private String headOfState;
        private City capital;
        private String code2;
    }
```

The City class is not created yet, let's go ahead and create it as follows:

```
@Data
@Setter
@Getter
public class City {
    private Long id;
    private String name;
    private Country country;
    private String district;
    private Long population;
}
```

Next is to model the CountryLanguage class, which is the language spoken in a country, as follows:

```
@Data
@Setter
@Getter
public class CountryLanguage {
    private Country country;
    private String language;
    private String isOfficial;
    private Double percentage;
}
```

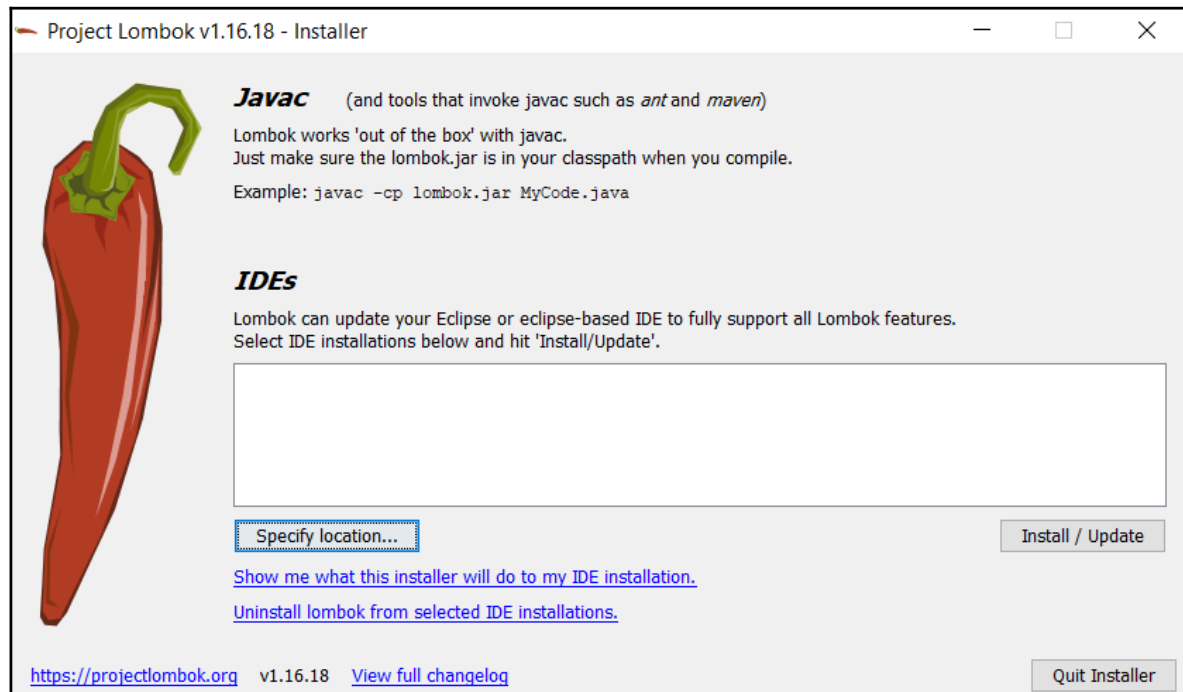
We also need a model class to map the GDP information obtained from the World Bank API. Let's go ahead and create a CountryGDP class as shown in the following code:

```
@Data
@Setter
@Getter
public class CountryGDP {
    private Short year;
    private Double value;
}
```

At this moment, everything works perfectly fine. But when you start calling getter and setter of these model classes into some other class, you may get a compilation error. This is because we need to do one more step to configure Lombok. After you defined the Maven dependency, you will see the JAR reference from IDE. Just right-click on it and select the **Run As | Java Application** option. Alternatively, you can execute the following command from terminal at the location where the Lombok JAR file is kept, as follows:

```
java -jar lombok-1.16.18.jar
```

Here, `lombok-1.16.18.jar` is the name of JAR file. You will see a separate window pop up as follows:



Select the location of your IDE by clicking on the **Specify location...** button. Once selected, click on the **Install / Update** button to install it. You will get a success message. Just restart the IDE and rebuild the project and you will see that just by defining `@Setter` and `@Getter`, the actual setters and getters are available to other classes. You are no longer required to add them explicitly.

Using Hibernate Validator to add validations

There are a few checks we need to add to our model classes so that the data being sent from the UI is not invalid. For this, we will make use of Hibernate Validator. You are required to add the Hibernate dependency as follows:

```
<properties>
  <java.version>1.8</java.version>
  <lombok.version>1.16.18</lombok.version>
</hibernate.validator.version>6.0.2.Final</hibernate.validator.version>
</properties>
<dependency>
  <groupId>org.hibernate.validator</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>${hibernate.validator.version}</version>
</dependency>
```

Now go back to `com.nilangpatel.worldgdp.model.Country` and update it with the following:

```
@Data public class Country {

    @NotNull @Size(max = 3, min = 3) private String code;
    @NotNull @Size(max = 52) private String name;
    @NotNull private String continent;
    @NotNull @Size(max = 26) private String region;
    @NotNull private Double surfaceArea;
    private Short indepYear;
    @NotNull private Long population;
    private Double lifeExpectancy;
    private Double gnp;
    @NotNull private String localName;
    @NotNull private String governmentForm;
    private String headOfState;
    private City capital;
    @NotNull private String code2;
}
```

Next is to update the `com.nilangpatel.worldgdp.model.City` class in a similar way, as follows:

```
@Data public class City {
    @NotNull private Long id;
    @NotNull @Size(max = 35) private String name;
    @NotNull @Size(max = 3, min = 3) private String countryCode;
    private Country country;
    @NotNull @Size(max = 20) private String district;
```



```
@NotNull private Long population;  
}
```

And finally, update `com.nilangpatel.worldgdp.model.CountryLanguage` class as well, as follows:

```
@Data  
public class CountryLanguage {  
    private Country country;  
    @NotNull private String countryCode;  
    @NotNull @Size(max = 30) private String language;  
    @NotNull @Size(max = 1, min = 1) private String isOfficial;  
    @NotNull private Double percentage;  
}
```

Defining the data access layer – Spring JDBC Template

We have the model classes that reflect the structure of the data in the database that we obtained from the World Bank API. Now we need to develop a data access layer that interacts with our MySQL and populates the data stored in the database into instances of the model classes. We will use the Spring JDBC Template to achieve the required interaction with the database.

First, we need the JDBC driver to connect any Java application with MySQL. This can be obtained by adding the following dependency and version property to our `pom.xml`:

```
<properties>  
    <java.version>1.8</java.version>  
    <lombok.version>1.16.18</lombok.version>  
<hibernate.validator.version>6.0.2.Final</hibernate.validator.version>  
    <mysql.jdbc.driver.version>5.1.44</mysql.jdbc.driver.version>  
</properties>  
<dependency>  
    <groupId>mysql</groupId>  
    <artifactId>mysql-connector-java</artifactId>  
    <version>${mysql.jdbc.driver.version}</version>  
</dependency>
```



Wherever you see

`<something.version>1.5.6</something.version>`, it should go within the `<properties></properties>` tag. Will not mention this repeatedly. This is for keeping the versions of libraries used in one place, making it easy to maintain and look up.

Anything that comes as `<dependency></dependency>` goes within the `<dependencies></dependencies>` list.

Now we need to add a dependency to the Spring core APIs, as well as the Spring JDBC APIs (which contain the JDBC Template) to our `pom.xml`. A brief intro about these two dependencies is as follows:

1. **Spring core APIs:** It provides us with core Spring features such as dependency injection and configuration model
2. **Spring JDBC APIs:** It provides us with the APIs required to create the `DataSource` instance and interact with the database



Since this is a sample application, we aren't using Hibernate or other ORM libraries because they provide lots of functionalities apart from basic CRUD operations. Instead, we will write SQL queries and use them with JDBC Template to make things simpler.

The following code shows the dependency information for the two libraries:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>${spring.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${spring.version}</version>
</dependency>
```

Along with the preceding two dependencies, we need to add a few more Spring dependencies to assist us in setting up Java-based configurations using annotations (such as `@bean`, `@Service`, `@Configuration`, `@ComponentScan`, and so on) and dependency injection using annotations (`@Autowired`). For this, we will be adding further dependencies as follows:

```
<dependency>
  <groupId>org.springframework</groupId>
```

```
<artifactId>spring-beans</artifactId>
<version>${spring.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>${spring.version}</version>
</dependency>
```

Defining the JDBC connection properties

We will define the JDBC connection properties in an `application.properties` file and place it in `src/main/resources`. The properties we define are as follows:

```
dataSourceClassName=com.mysql.jdbc.Driver
jdbcUrl=jdbc:mysql://localhost:3306/worldgdp
dataSource.user=root
dataSource.password=test
```

The preceding properties are with the assumptions that MySQL is running on port 3306 and the database username and password are `root` and `test` respectively. You can change these properties as per your local configuration. The next step is to define a properties resolver that will be able to resolve the properties when used from within the code. We will use the `@PropertySource` annotation, along with an instance of `PropertySourcesPlaceholderConfigurer`, as shown in the following code:

```
@Configuration
@PropertySource("classpath:application.properties")
public class PropertiesWithJavaConfig {

    @Bean
    public static PropertySourcesPlaceholderConfigurer
        propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }
}
```



We will follow the convention of placing all our configuration classes in `com.nilangpatel.worldgdp.config` and any root configuration will go in the `com.nilangpatel.worldgdp` package.

This class reads all the properties from the `application.properties` file stored in classpath (`src/main/resources`). Next up is to configure a `javax.sql.DataSource` object that will connect to the database using the properties defined in the `application.properties` file. We will use the HikariCP connection pooling library for creating our `DataSource` instance. This `DataSource` instance is then used to instantiate `NamedParameterJdbcTemplate`. We will use `NamedParameterJdbcTemplate` to execute all our SQL queries. At this point, we need to add a necessary dependency for the HikariCP library as follows:

```
<dependency>
  <groupId>com.zaxxer</groupId>
  <artifactId>HikariCP</artifactId>
  <version>${hikari.version}</version>
</dependency>
```

The `DBConfiguration` data source configuration class should look as follows:

```
@Configuration
public class DBConfiguration {
    @Value("${jdbcUrl}") String jdbcUrl;
    @Value("${dataSource.user}") String username;
    @Value("${dataSource.password}") String password;
    @Value("${dataSource.className}") String className;
    @Bean
    public DataSource getDataSource() {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl(jdbcUrl);
        ds.setUsername(username);
        ds.setPassword(password);
        ds.setDriverClassName(className);
        return ds;
    }
    @Bean
    public NamedParameterJdbcTemplate namedParamJdbcTemplate() {
        NamedParameterJdbcTemplate namedParamJdbcTemplate =
            new NamedParameterJdbcTemplate(getDataSource());
        return namedParamJdbcTemplate;
    }
}
```

Let's have a quick introduction to a few new things used in this code:

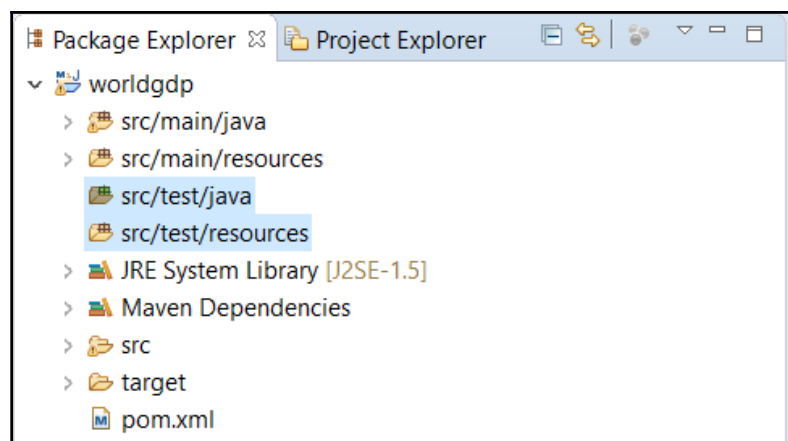
- **@Configuration:** This is to indicate to Spring Framework that this class creates Java objects that contain some configuration
- **@Bean:** This is method-level annotation, used to indicate to Spring Framework that the method returns Java objects whose life cycle is managed by Spring Framework and injected into places where its dependency is declared
- **@Value:** This is used to refer to the properties defined in the `application.properties`, which are resolved by the `PropertySourcesPlaceholderConfigurer` bean defined in the `PropertiesWithJavaConfig` class

It is always good practice to write unit test cases in JUnit. We will write test cases for our application. For that, we need to create the corresponding configuration classes for running our JUnit tests. In the next section, we will look at setting up the test environment.

Setting up the test environment

Let's adopt a test first approach here. So, before going into writing the queries and DAO classes, let's set up the environment for our unit testing. If you don't find the `src/test/java` and `src/test/resources` folders, then please go ahead and create them either from your IDE or from your OS file explorer.

The `src/test/java` folder will contain all the Java code and `src/test/resources` will contain the required property files and other resources required for test cases. After creating the required folders, the project structure looks something like that shown in the following screenshot:



We will use the H2 database as a source of data for our testing environment. For that, we will update our Maven dependencies to add H2 and JUnit dependencies. H2 is one of the most popular embedded databases. The following is the dependency information that you need to add in your `pom.xml`:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-test</artifactId>
  <version>${spring.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.assertj</groupId>
  <artifactId>assertj-core</artifactId>
  <version>${assertj.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <version>${h2.version}</version>
</dependency>
```

We already have a property for `spring.version`, but we need version properties for the other two, as given in the following code:

```
<junit.version>4.12</junit.version>
<assertj.version>3.12.0</assertj.version>
<h2.version>1.4.198</h2.version>
```

The World DB schema available in MySQL will not be compatible to run with H2, but don't worry. The compatible World DB schema for H2 is available in the source code of this chapter, you can download from GitHub (<https://github.com/PacktPublishing/Spring-5.0-Projects/tree/master/chapter01>). It is kept in the `src/test/resources` folder in the project. The file name is `h2_world.sql`. We will use this file to bootstrap our H2 database with the required tables and data that will then be available in our tests.

Next up is to configure H2 and one of the things we configure is the name of the SQL script file that contains the schema and data. This SQL script file should be available on the classpath. The following is the configuration class created in the `com.nilangpatel.worldgdp.test.config` package under `src/test/java` folder:

```
@Configuration
public class TestDBConfiguration {
    @Bean
    public DataSource dataSource() {
        return new EmbeddedDatabaseBuilder()
            .generateUniqueName(true)
            .setType(EmbeddedDatabaseType.H2)
            .setScriptEncoding("UTF-8")
            .ignoreFailedDrops(true)
            .addScript("h2_world.sql")
            .build();
    }
    @Bean("testTemplate")
    public NamedParameterJdbcTemplate namedParamJdbcTemplate() {
        NamedParameterJdbcTemplate namedParamJdbcTemplate =
            new NamedParameterJdbcTemplate(dataSource());
        return namedParamJdbcTemplate;
    }
}
```

Along with the H2 configuration, we are initializing `NamedParameterJdbcTemplate` by providing it with the H2 datasource built in the other method.



We have added few other dependencies specific to JUnit. You can refer to them by downloading the source code.

Defining the RowMapper

As we are using the JDBC Template, we need a way to map the rows of data from a database to a Java object. This can be achieved by implementing a `RowMapper` interface. We will define mapper classes for all the three entities. For `Country`, the raw mapper class looks as follows:

```
public class CountryRowMapper implements RowMapper<Country>{

    public Country mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        Country country = new Country();
    }
}
```

```

country.setCode(rs.getString("code"));
country.setName(rs.getString("name"));
country.setContinent(rs.getString("continent"));
country.setRegion(rs.getString("region"));
country.setSurfaceArea(rs.getDouble("surface_area"));
country.setIndepYear(rs.getShort("indep_year"));
country.setPopulation(rs.getLong("population"));
country.setLifeExpectancy(rs.getDouble("life_expectancy"));
country.setGnp(rs.getDouble("gnp"));
country.setLocalName(rs.getString("local_name"));
country.setGovernmentForm(rs.getString("government_form"));
country.setHeadOfState(rs.getString("head_of_state"));
country.setCode2(rs.getString("code2"));
if ( Long.valueOf(rs.getLong("capital")) != null ) {
    City city = new City();
    city.setId(rs.getLong("capital"));
    city.setName(rs.getString("capital_name"));
    country.setCapital(city);
}
return country;
}
}

```

Then we define the mapper class for City as follows:

```

public class CityRowMapper implements RowMapper<City>{
    public City mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        City city = new City();
        city.setCountryCode(rs.getString("country_code"));
        city.setDistrict(rs.getString("district"));
        city.setId(rs.getLong("id"));
        city.setName(rs.getString("name"));
        city.setPopulation(rs.getLong("population"));
        return city;
    }
}

```

And finally, we define CountryLanguage as follows:

```

public class CountryLanguageRowMapper implements
    RowMapper<CountryLanguage> {
    public CountryLanguage mapRow(ResultSet rs, int rowNum)
        throws SQLException {
        CountryLanguage countryLng = new CountryLanguage();
        countryLng.setCountryCode(rs.getString("countrycode"));
        countryLng.setIsOfficial(rs.getString("isofficial"));
        countryLng.setLanguage(rs.getString("language"));
    }
}

```



```

        countryLng.setPercentage(rs.getDouble("percentage"));
        return countryLng;
    }
}

```

Designing the CountryDAO

Let's go ahead and define the CountryDAO class in the `com.nilangpatel.worldgdp.dao` package along with the required methods, starting with the `getCountries` method. This method will fetch the details of countries to show them in the listing page. This method is also called while filtering the country list. Based on listing, filtering, and paginating, we have broken up the query used in this method into the following parts:

1. Select clause:

```

private static final String SELECT_CLAUSE = "SELECT "
    + " c.Code, "
    + " c.Name, "
    + " c.Continent, "
    + " c.region, "
    + " c.SurfaceArea surface_area, "
    + " c.IndepYear indep_year, "
    + " c.Population, "
    + " c.LifeExpectancy life_expectancy, "
    + " c.GNP, "
    + " c.LocalName local_name, "
    + " c.GovernmentForm government_form, "
    + " c.HeadOfState head_of_state, "
    + " c.code2 , "
    + " c.capital , "
    + " cy.name capital_name "
    + " FROM country c"
    + " LEFT OUTER JOIN city cy ON cy.id = c.capital ";

```

2. Search where clause:

```

private static final String SEARCH_WHERE_CLAUSE = " AND (
    LOWER(c.name) "
    + " LIKE CONCAT('%', LOWER(:search), '%') ) ";

```

3. Continent filter where clause:

```

private static final String CONTINENT_WHERE_CLAUSE =
    " AND c.continent = :continent ";

```

4. Region filter where clause:

```
private static final String REGION_WHERE_CLAUSE =
    " AND c.region = :region ";
```

5. Pagination clause:

```
private static final String PAGINATION_CLAUSE = " ORDER BY c.code "
    + " LIMIT :size OFFSET :offset ";
```

The placeholders defined by `:<<variableName>>` are replaced by the values provided in the Map to the `NamedParameterJdbcTemplate`. This way we can avoid concatenating the values into the SQL query, thereby avoiding chances of SQL injection.

The `getCountries()` definition would now be as follows:

```
public List<Country> getCountries(Map<String, Object> params){
    int pageNo = 1;
    if ( params.containsKey("pageNo") ) {
        pageNo = Integer.parseInt(params.get("pageNo").toString());
    }
    Integer offset = (pageNo - 1) * PAGE_SIZE;
    params.put("offset", offset);
    params.put("size", PAGE_SIZE);
    return namedParamJdbcTemplate.query(SELECT_CLAUSE
        + " WHERE 1 = 1 "
        + (!StringUtils.isEmpty((String)params.get("search"))
            ? SEARCH_WHERE_CLAUSE : "")
        + (!StringUtils.isEmpty((String)params.get("continent"))
            ? CONTINENT_WHERE_CLAUSE : "")
        + (!StringUtils.isEmpty((String)params.get("region"))
            ? REGION_WHERE_CLAUSE : "")
        + PAGINATION_CLAUSE,
        params, new CountryRowMapper());
}
```

Next is to implement the `getCountriesCount` method, which is similar to `getCountries`, except that it returns the count of entries matching the `WHERE` clause without the pagination applied. The implementation is as shown in the following code:

```
public int getCountriesCount(Map<String, Object> params) {
    return namedParamJdbcTemplate.queryForObject(
        "SELECT COUNT(*) FROM country c"
        + " WHERE 1 = 1 "
        + (!StringUtils.isEmpty((String)params.get("search"))
            ? SEARCH_WHERE_CLAUSE : "")
        + (!StringUtils.isEmpty((String)params.get("continent"))
            ? CONTINENT_WHERE_CLAUSE : ""));
}
```

```
+ (!StringUtils.isEmpty((String)params.get("region"))
    ? REGION_WHERE_CLAUSE : ""),
    params, Integer.class);
}
```

Then we implement the `getCountryDetail` method to get the detail of the country, given its code, as follows:

```
public Country getCountryDetail(String code) {
    Map<String, String> params = new HashMap<String, String>();
    params.put("code", code);
    return namedParamJdbcTemplate.queryForObject(SELECT_CLAUSE
        +" WHERE c.code = :code", params,
        new CountryRowMapper());
}
```



In all of the previous DAO method implementations, we have made use of the `CountryRowMapper` we defined in the *Defining the RowMapper* section.

Finally, we define the method to allow editing the country information, as shown in the following code:

```
public void editCountryDetail(String code, Country country) {
    namedParamJdbcTemplate.update(" UPDATE country SET "
        + " name = :name, "
        + " localname = :localName, "
        + " capital = :capital, "
        + " continent = :continent, "
        + " region = :region, "
        + " HeadOfState = :headOfState, "
        + " GovernmentForm = :governmentForm, "
        + " IndepYear = :indepYear, "
        + " SurfaceArea = :surfaceArea, "
        + " population = :population, "
        + " LifeExpectancy = :lifeExpectancy "
        + "WHERE Code = :code ",
        getCountryAsMap(code, country));
}
```

The previous method uses a helper method that builds a `Map` object, by using the data present in the `Country` object. We need the map, as we'll be using it as a parameter source for our `namedParamJdbcTemplate`.

The helper method has a simple implementation, as shown in the following code:

```
private Map<String, Object> getCountryAsMap(String code, Country country){
    Map<String, Object> countryMap = new HashMap<String, Object>();
    countryMap.put("name", country.getName());
    countryMap.put("localName", country.getLocalName());
    countryMap.put("capital", country.getCapital().getId());
    countryMap.put("continent", country.getContinent());
    countryMap.put("region", country.getRegion());
    countryMap.put("headOfState", country.getHeadOfState());
    countryMap.put("governmentForm", country.getGovernmentForm());
    countryMap.put("indepYear", country.getIndepYear());
    countryMap.put("surfaceArea", country.getSurfaceArea());
    countryMap.put("population", country.getPopulation());
    countryMap.put("lifeExpectancy", country.getLifeExpectancy());
    countryMap.put("code", code);
    return countryMap;
}
```

Let's write our JUnit test for the CountryDAO class, which we haven't created yet.

Create CountryDAOTest class into the com.nilangpatel.worldgdp.test.dao package as follows:

```
@RunWith(SpringRunner.class)
@SpringJUnitConfig( classes = {
    TestDBConfiguration.class, CountryDAO.class})
public class CountryDAOTest {

    @Autowired CountryDAO countryDao;
    @Autowired @Qualifier("testTemplate")
    NamedParameterJdbcTemplate namedParamJdbcTemplate;
    @Before
    public void setup() {
        countryDao.setNamedParamJdbcTemplate(namedParamJdbcTemplate);
    }
    @Test
    public void testGetCountries() {
        List<Country> countries = countryDao.getCountries(new HashMap<>());
        //AssertJ assertions
        //Paginated List, so should have 20 entries
        assertThat(countries).hasSize(20);
    }
    @Test
    public void testGetCountries_searchByName() {
        Map<String, Object> params = new HashMap<>();
        params.put("search", "Aruba");
        List<Country> countries = countryDao.getCountries(params);
        assertThat(countries).hasSize(1);
    }
}
```

```

    }
    @Test
    public void testGetCountries_searchByContinent() {
        Map<String, Object> params = new HashMap<>();
        params.put("continent", "Asia");
        List<Country> countries = countryDao.getCountries(params);
        assertThat(countries).hasSize(20);
    }
    @Test
    public void testGetCountryDetail() {
        Country c = countryDao.getCountryDetail("IND");
        assertThat(c).isNotNull();
        assertThat(c.toString()).isEqualTo("Country(code=IND, name=India, "
            + "continent=Asia, region=Southern and Central Asia, "
            + "surfaceArea=3287263.0, indepYear=1947, population=1013662000, "
            + "lifeExpectancy=62.5, gnp=447114.0, localName=Bharat/India, "
            + "governmentForm=Federal Republic, headOfState=Kocheril Raman
Narayanan, "
            + "capital=City(id=1109, name=New Delhi, countryCode=null, "
            + "country=null, district=null, population=null), code2=IN)");
    }
    @Test public void testEditCountryDetail() {
        Country c = countryDao.getCountryDetail("IND");
        c.setHeadOfState("Ram Nath Kovind");
        c.setPopulation(13241713541);
        countryDao.editCountryDetail("IND", c);
        c = countryDao.getCountryDetail("IND");
        assertThat(c.getHeadOfState()).isEqualTo("Ram Nath Kovind");
        assertThat(c.getPopulation()).isEqualTo(13241713541);
    }
    @Test public void testGetCountriesCount() {
        Integer count = countryDao.getCountriesCount(Collections.EMPTY_MAP);
        assertThat(count).isEqualTo(239);
    }
}

```

There are a few things to note about configuring JUnit tests using the Spring test framework from the following test, including the following:

- `@RunWith` is used to replace the JUnit's test runner with a custom test runner, which in this case, is Spring's `SpringRunner`. Spring's test runner helps in integrating JUnit with the Spring test framework.
- `@SpringJUnitConfig` is used to provide the list of classes that contain the required configuration to satisfy the dependencies for running the test.



Many people who choose ORM frameworks may feel that writing complicated SQL queries like this is awkward. However, from the next chapter onward, we'll start using the Spring Data framework to make an interaction with various data sources; the database is one of those accessed with the Spring Data JPA. Here, we wanted to show how the Spring JDBC offering interacts with the database.

Designing the CityDAO

The following are some of the important operations to be supported by `com.nilangpatel.worldgdp.dao.CityDAO` class:

- Get cities for a country
- Get city details for given ID
- Add a new city to a country
- Delete the given city from the country

Let's go ahead and implement each one of these functionalities starting with the `getCities`, as follows:

```
public List<City> getCities(String countryCode, Integer pageNo){
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("code", countryCode);
    if ( pageNo != null ) {
        Integer offset = (pageNo - 1) * PAGE_SIZE;
        params.put("offset", offset);
        params.put("size", PAGE_SIZE);
    }
    return namedParamJdbcTemplate.query("SELECT "
        + " id, name, countrycode country_code, district, population "
        + " FROM city WHERE countrycode = :code"
        + " ORDER BY Population DESC"
        + ((pageNo != null) ? " LIMIT :offset , :size " : ""),
        params, new CityRowMapper());
}
```

We are using a paginated query to get a list of cities for a country. We will also need another overloaded version of this method where we return all the cities of a country and we will use this query to fetch all the cities while editing the country to select its capital. The overloaded version is as follows:

```
public List<City> getCities(String countryCode){
    return getCities(countryCode, null);
}
```

Next is to implement the method to get the city details, as shown in the following code:

```
public City getCityDetail(Long cityId) {
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("id", cityId);
    return namedParamJdbcTemplate.queryForObject("SELECT id, "
        + " name, countrycode country_code, "
        + " district, population "
        + " FROM city WHERE id = :id",
        params, new CityRowMapper());
}
```

Then we implement the method to add a city as follows:

```
public Long addCity(String countryCode, City city) {
    SqlParameterSource paramSource = new MapSqlParameterSource(
        getMapForCity(countryCode, city));
    KeyHolder keyHolder = new GeneratedKeyHolder();
    namedParamJdbcTemplate.update("INSERT INTO city("
        + " name, countrycode, "
        + " district, population) "
        + " VALUES (:name, :country_code, "
        + " :district, :population )",
        paramSource, keyHolder);
    return keyHolder.getKey().longValue();
}
```

As we saw with adding a country, this will also make use of a helper method to return a Map from the City data, as follows:

```
private Map<String, Object> getMapForCity(String countryCode, City city){
    Map<String, Object> map = new HashMap<String, Object>();
    map.put("name", city.getName());
    map.put("country_code", countryCode);
    map.put("district", city.getDistrict());
    map.put("population", city.getPopulation());
    return map;
}
```

An important thing to notice in `addCity` is the use of `KeyHolder` and `GeneratedKeyHolder` to return the generated (due to auto increment) primary key that is the `cityId`, as follows:

```
KeyHolder keyHolder = new GeneratedKeyHolder();
//other code
return keyHolder.getKey().longValue();
```

And finally, we implement the method to delete a city from the country as shown in the following code:

```
public void deleteCity(Long cityId) {
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("id", cityId);
    namedParamJdbcTemplate.update("DELETE FROM city WHERE id = :id", params);
}
```

Now let's add a test for CityDAO. Add the CityDAOTest class in com.nilangpatel.worldgdp.test.dao package under src/test/java folder as follows:

```
@RunWith(SpringRunner.class)
@SpringJUnitConfig(classes = {
    TestDBConfiguration.class, CityDAO.class})
public class CityDAOTest {

    @Autowired CityDAO cityDao;
    @Autowired @Qualifier("testTemplate")
    NamedParameterJdbcTemplate namedParamJdbcTemplate;
    @Before
    public void setup() {
        cityDao.setNamedParamJdbcTemplate(namedParamJdbcTemplate);
    }
    @Test public void testGetCities() {
        List<City> cities = cityDao.getCities("IND", 1);
        assertThat(cities).hasSize(10);
    }
    @Test public void testGetCityDetail() {
        Long cityId = 10241;
        City city = cityDao.getCityDetail(cityId);
        assertThat(city.toString()).isEqualTo("City(id=1024, name=Mumbai
(Bombay), "
        + "countryCode=IND, country=null, district=Maharashtra,
population=10500000)");
    }
    @Test public void testAddCity() {
        String countryCode = "IND";
        City city = new City();
        city.setCountryCode(countryCode);
        city.setDistrict("District");
        city.setName("City Name");
        city.setPopulation(1010101);
        long cityId = cityDao.addCity(countryCode, city);
        assertThat(cityId).isNotNull();
        City cityFromDb = cityDao.getCityDetail(cityId);
        assertThat(cityFromDb).isNotNull();
    }
}
```



```

        assertThat(cityFromDb.getName()).isEqualTo("City Name");
    }
    @Test (expected = EmptyResultDataAccessException.class)
    public void testDeleteCity() {
        Long cityId = addCity();
        cityDao.deleteCity(cityId);
        City cityFromDb = cityDao.getCityDetail(cityId);
        assertThat(cityFromDb).isNull();
    }
    private Long addCity() {
        String countryCode = "IND";
        City city = new City();
        city.setCountryCode(countryCode);
        city.setDistrict("District");
        city.setName("City Name");
        city.setPopulation(1010101);
        return cityDao.addCity(countryCode, city);
    }
}

```

Designing the CountryLanguageDAO

We will need to expose the following APIs to interact with the `countrylanguage` table:

- Get list of languages for a given country code
- Add a new language for a country by checking that the language doesn't already exist
- Delete a language for a country

For the sake of keeping it short, we will show the method implementations covering these three scenarios. The complete code can be found in the

`com.nilangpatel.worldgdp.dao.CountryLanguageDAO` class available in the code downloaded for this book. The following is the code for these method implementations:

```

public List<CountryLanguage> getLanguages(String countryCode, Integer
pageNo){
    Map<String, Object> params = new HashMap<String, Object>();
    params.put("code", countryCode);
    Integer offset = (pageNo - 1) * PAGE_SIZE;
    params.put("offset", offset);
    params.put("size", PAGE_SIZE);
    return namedParamJdbcTemplate.query("SELECT * FROM countrylanguage"
        + " WHERE countrycode = :code"
        + " ORDER BY percentage DESC "
        + " LIMIT :size OFFSET :offset ",

```

```
        params, new CountryLanguageRowMapper());
    }

    public void addLanguage(String countryCode, CountryLanguage cl) {
        namedParamJdbcTemplate.update("INSERT INTO countrylanguage ( "
            + " countrycode, language, isofficial, percentage ) "
            + " VALUES ( :country_code, :language, "
            + " :is_official, :percentage ) ",
            getAsMap(countryCode, cl));
    }

    public boolean languageExists(String countryCode, String language) {
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("code", countryCode);
        params.put("lang", language);
        Integer langCount = namedParamJdbcTemplate.queryForObject(
            "SELECT COUNT(*) FROM countrylanguage"
            + " WHERE countrycode = :code "
            + " AND language = :lang", params, Integer.class);
        return langCount > 0;
    }

    public void deleteLanguage (String countryCode, String language ) {
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("code", countryCode);
        params.put("lang", language);
        namedParamJdbcTemplate.update("DELETE FROM countrylanguage "
            + " WHERE countrycode = :code AND "
            + " language = :lang ", params);
    }

    private Map<String, Object> getAsMap(String countryCode, CountryLanguage
cl){
        Map<String, Object> map = new HashMap<String, Object>();
        map.put("country_code", countryCode);
        map.put("language", cl.getLanguage());
        map.put("is_official", cl.getIsOfficial());
        map.put("percentage", cl.getPercentage());
        return map;
    }
}
```

Designing the client for World Bank API

We need to fetch the GDP data from WorldBank API. As we discussed, it is REST end point, where we have to send few parameters and will get the response. For this, we will use RestTemplate to make REST call. The following is the definition for the `com.packt.external.WorldBankApiClient` class, which is used to invoke the World Bank API and process its response to return `List<CountryGDP>`:

```
@Service
public class WorldBankApiClient {

    String GDP_URL =
        "http://api.worldbank.org/countries/%s/indicators/NY.GDP.MKTP.CD?"
        + "format=json&date=2008:2018";

    public List<CountryGDP> getGDP(String countryCode) throws ParseException
    {
        RestTemplate worldBankRestTplt = new RestTemplate();
        ResponseEntity<String> response
            = worldBankRestTplt.getForEntity(String.format(GDP_URL,
countryCode), String.class);
        //the second element is the actual data and its an array of object
        JSONParser parser = new JSONParser();
        JSONArray responseData = (JSONArray) parser.parse(response.getBody());
        JSONArray countryDataArr = (JSONArray) responseData.get(1);
        List<CountryGDP> data = new ArrayList<CountryGDP>();
        JSONObject countryDataYearWise=null;
        for (int index=0; index < countryDataArr.size(); index++) {
            countryDataYearWise = (JSONObject) countryDataArr.get(index);
            String valueStr = "0";
            if(countryDataYearWise.get("value") !=null) {
                valueStr = countryDataYearWise.get("value").toString();
            }
            String yearStr = countryDataYearWise.get("date").toString();
            CountryGDP gdp = new CountryGDP();
            gdp.setValue(valueStr != null ? Double.valueOf(valueStr) : null);
            gdp.setYear(Short.valueOf(yearStr));
            data.add(gdp);
        }
        return data;
    }
}
```

Defining the API controllers

So far, we have written code to interact with the DB. Next up is to work on the code for the controller. We will have both types of controller—one that returns the view name (Thymeleaf template in our case) with the data for the view populated in the model object, and the other that exposes the RESTful APIs. We will need to add the following dependency to `pom.xml`:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>${spring.version}</version>
</dependency>
```



Adding `spring-webmvc` to the dependency will automatically include `spring-core`, `spring-beans`, and `spring-context` dependencies. So we can remove them from the `pom.xml`.

Enabling Web MVC using `@EnableWebMvc`

To be able to make use of the Spring MVC features, we need to have one class that has been annotated with `@Configuration`, to be annotated with `@EnableWebMvc`.

The `@EnableWebMvc` annotation, imports the Spring MVC configuration from the `WebMvcConfigurationSupport` class present in the Spring MVC framework. If we need to override any of the default imported configuration, we would have to implement the `WebMvcConfigurer` interface present in the Spring MVC framework and override the required methods.

We will create an `AppConfiguration` class with the following definition:

```
@EnableWebMvc
@Configuration
@ComponentScan(basePackages = "com.nilangpatel.worldgdp")
public class AppConfiguration implements WebMvcConfigurer{

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/static/**").addResourceLocations("/static/");
    }
}
```

In the previous configuration, a few important things to note are as follows:

- `@EnableWebMvc`: This imports the Spring MVC related configuration.
- `@ComponentScan`: This is used for declaring the packages that have to be scanned for Spring components (which can be `@Configuration`, `@Service`, `@Controller`, `@Component`, and so on). If no package is defined, then it scans starting from the package where the class is defined.
- `WebMvcConfigurer`: We are going to implement this interface to override some of the default Spring MVC configuration seen in the previous code.

Configuration to deploy to Tomcat without web.xml

As we will be deploying the application to Tomcat, we need to provide the servlet configuration to the application server. We will look at how to deploy to Tomcat in a separate section, but now we will look at the Java configuration, which is sufficient to deploy the application to Tomcat or any application server without the need for an additional `web.xml`. The Java class definition is given in the following:

```
public class WorldApplicationInitializer extends
    AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return null;
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] {AppConfiguration.class};
    }
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

The `AbstractAnnotationConfigDispatcherServletInitializer` abstract class is an implementation of the `WebApplicationInitializer` interface that is used to register Spring's `DispatcherServlet` instance and uses the other `@Configuration` classes to configure the `DispatcherServlet`.

We just need to override the `getRootConfigClasses()`, `getServletConfigClasses()`, and `getServletMappings()` methods. The first two methods point to the configuration classes that need to load into the servlet context, and the last method is used to provide the servlet mapping for `DispatcherServlet`.

`DispatcherServlet` follows the front controller pattern, where there is a single servlet registered to handle all the web requests. This servlet uses the `RequestMapping` and invokes the corresponding implementation based on the URL mapped to the implementation.

We need to make a small update to the Maven WAR plugin so that it doesn't fail if there is no `web.xml` found. This can be done by updating the `<plugins>` tag in the `pom.xml` file, as shown in the following:

```
<build>
  <finalName>worldgdp</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <executions>
        <execution>
          <id>default-war</id>
          <phase>prepare-package</phase>
          <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Now we are all set to implement our controllers. We will show you how to deploy to Tomcat once we have implemented all the RESTful API controllers.

Defining the RESTful API controller for country resource

Let's define the RESTful API controller for the country resource. The following is the template for the controller:

```
@RestController
@RequestMapping("/api/countries")
@Slf4j
public class CountryApiController {
    @Autowired CountryDAO countryDao;
    @Autowired WorldBankApiClient worldBankApiClient;
    @GetMapping
    public ResponseEntity<?> getCountries(
        @RequestParam(name="search", required = false) String searchTerm,
        @RequestParam(name="continent", required = false) String continent,
        @RequestParam(name="region", required = false) String region,
        @RequestParam(name="pageNo", required = false) Integer pageNo
    ){
        //logic to fetch contries from CountryDAO
        return ResponseEntity.ok();
    }
    @PostMapping(value =("/{countryCode}",
        consumes = {MediaType.APPLICATION_JSON_VALUE})
    public ResponseEntity<?> editCountry(
        @PathVariable String countryCode, @Valid @RequestBody Country country
    ){
        //logic to edit existing country
        return ResponseEntity.ok();
    }
    @GetMapping("/{countryCode}/gdp")
    public ResponseEntity<?> getGDP(@PathVariable String countryCode){
        //logic to get GDP by using external client
        return ResponseEntity.ok();
    }
}
```

The following are a few things to note from the previous code:

- **@RestController:** This is used to annotate a class as a controller with each of the RESTful methods returning the data in the response body.
- **@RequestMapping:** This is for assigning the root URL for accessing the resources.

- **@GetMapping and @PostMapping:** These are used to assign the HTTP verbs that will be used to invoke the resources. The URL for the resources are passed within the annotation, along with other request headers that consume and produce information.

Let's implement each of the methods in order, starting with `getCountries()`, as shown in the following code:

```
@GetMapping
public ResponseEntity<?> getCountries(
    @RequestParam(name="search", required = false) String searchTerm,
    @RequestParam(name="continent", required = false) String continent,
    @RequestParam(name="region", required = false) String region,
    @RequestParam(name="pageNo", required = false) Integer pageNo
){
    try {
        Map<String, Object> params = new HashMap<String, Object>();
        params.put("search", searchTerm);
        params.put("continent", continent);
        params.put("region", region);
        if ( pageNo != null ) {
            params.put("pageNo", pageNo.toString());
        }
        List<Country> countries = countryDao.getCountries(params);
        Map<String, Object> response = new HashMap<String, Object>();
        response.put("list", countries);
        response.put("count", countryDao.getCountriesCount(params));
        return ResponseEntity.ok(response);
    } catch (Exception ex) {
        log.error("Error while getting countries", ex);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body("Error while getting countries");
    }
}
```

The following are some of the things to note from the previous code:

- **@RequestParam:** This annotation is used to declare request parameters accepted by the controller endpoint. The parameters can be provided with a default value and can also be made mandatory.
- **ResponseEntity:** This class is used to return the response body, along with other response parameters such as status, headers, and so on.

Next up is the API for editing country details, as follows:

```
@PostMapping("/{countryCode}")
public ResponseEntity<?> editCountry(
    @PathVariable String countryCode, @Valid @RequestBody Country country ){
    try {
        countryDao.editCountryDetail(countryCode, country);
        Country countryFromDb = countryDao.getCountryDetail(countryCode);
        return ResponseEntity.ok(countryFromDb);
    }catch(Exception ex) {
        log.error("Error while editing the country: {} with data: {}",
            countryCode, country, ex);
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
            .body("Error while editing the country");
    }
}
```

The following are a few things to note from the previous code implementation:

- **@PathVariable**: This is used to declare any variable that needs to be part of the URL path of the controller endpoint. In our case, we want the country code to be part of the URL. So the URL will be of the `/api/countries/IND` form.
- **@Valid**: This triggers the Bean Validation API to check for the restrictions on each of the class properties. If the data from the client is not valid, it returns a **400**.
- **@RequestBody**: This is used to capture the data sent in the request body and the Jackson library is used to convert the JSON data in the request body to the corresponding Java object.

The rest of the API implementation can be found in the `CountryAPIController` class. The tests for the API controller can be found in the `CountryAPIControllerTest` class, which is available in the source code of this book.

Defining the RESTful API controller for city resource

For the city resource we would need the following APIs:

- Get cities for a given country
- Add a new city to the country
- Delete the city from the country

The code for this controller can be found in the `CityAPIController` class and the tests for the API controller can be found in the `CityAPIControllerTest` class, which is available in the source code of this book.

Defining the RESTful API controller for country language resource

For the `CountryLanguage` resource we need the following APIs:

- Get languages for a country
- Add a language for a country
- Delete a language from the country

The code for this controller can be found in the `CountryLanguageAPIController` class and the tests for the API controller can be found in the `CountryLanguageAPIControllerTest` class, which is available in the source code of this book.

Deploying to Tomcat

Before we proceed with View and Controller for handling views, we will deploy the app developed so far to Tomcat. You can download Tomcat 8.5 from here (<https://tomcat.apache.org/download-80.cgi>). Installation is as simple as extracting the ZIP/TAR file onto your file system.

Let's create a user `admin` and `manager-gui` role in Tomcat. To do this, have to edit `apache-tomcat-8.5.23/conf/tomcat-users.xml` and add the following line:

```
<role rolename="manager-gui" />
<user username="admin" password="admin" roles="manager-gui" />
```

Starting up Tomcat is simple, as follows:

1. Navigate to `apache-tomcat-8.5.23/bin`
2. Run `startup.bat`

Navigate to `http://localhost:8080/manager/html` and enter `admin`, and `admin` for username and password respectively, to be able to view Tomcat's manager console. The initial part of the page will list the applications deployed in the current instance, and toward the later part of the page you will find an option to upload a WAR file to deploy the application, as shown in the following screenshot:

We can either upload the WAR file generated after running `mvn package` or update the `server.xml` of the Tomcat instance to refer to the target directory of the project to be able to deploy automatically. The latter approach can be used for development, while the former that is WAR deployment can be used for production.

In a production system, you can have a continuous deployment server generate a WAR file and deploy to a remote Tomcat instance. In this scenario, we will use the latter approach of updating the Tomcat's configuration. You have to add the following line of code in the Tomcat's `server.xml` file, located at `TOMCAT_HOME/conf/server.xml`:

```
<Context path="/world" docBase="<<Directory path where you keep WAR file>>"
      reloadable="true" />
```

The preceding line has to be added between the `<Host></Host>` tags. Alternatively, you can configure Tomcat in your IDE (for example, Eclipse), which is more convenient for development purposes. We will build the project with Maven, but before that, please add following configuration to the `<properties></properties>` section of `pom.xml`:

```
<maven.compiler.target>1.8</maven.compiler.target>
<maven.compiler.source>1.8</maven.compiler.source>
```

This will make sure to choose the correct Java compiler version while building (packaging) the application with Maven from the command line. Next is to build the project using the `mvn package` and run Tomcat using `TOMCAT_HOME/bin/startup.bat`, and once the server is UP, you can visit the API `http://localhost:8080/worldgdp/api/countries` in the browser to see the following input:

```
{
  "count": 239,
  "list": [
    { ... }, // 14 items
    { ... }, // 14 items
    {
      "code": "AGO",
      "name": "Angola",
      "continent": "Africa",
      "region": "Central Africa",
      "surfaceArea": 1246700,
      "indepYear": 1975,
      "population": 12878000,
      "lifeExpectancy": 38.29999923706055,
      "gnp": 6648,
      "localName": "Angola",
      "governmentForm": "Republic",
      "headOfState": "Jos  Eduardo dos Santos",
      "capital": {
        "id": 56,
        "name": "Luanda",
        "countryCode": null,
        "country": null,
        "district": null,
        "population": null
      },
      "code2": "AO"
    },
    {
      "code": "AIA",
      "name": "Anguilla",
      "continent": "North America",
```

Defining the view controller

We will have one view controller, `ViewController.java` defined in the `com.nilangpatel.worldgdp.controller.view`. The view controller will be responsible for populating the data required for the view templates and also mapping URLs to corresponding view templates.

We will be using Thymeleaf (www.thymeleaf.org) as the server-side template engine and Mustache.js (<https://github.com/janl/mustache.js>) as our client-side template engine. The advantage of using a client-side template engine is that any data loaded asynchronously in the form of JSON can easily be added to the DOM by generating HTML using the client-side templates. We will explore more about Thymeleaf and Mustache.js in Chapter 3, *Blogpress – A simple blog management system*.

There are much better ways to do this by using frameworks such as Vue.js, React.js, Angular.js, and so on. We will look at the view template in the next section. Let's continue our discussion about the view controller. The view controller should map the right view template and the data for the following scenarios:

- Listing of countries
- Viewing country detail
- Editing country detail

Let's look at the following skeletal structural definition of the `ViewController` class:

```
@Controller
@RequestMapping("/")
public class ViewController {
    @Autowired CountryDAO countryDao;
    @Autowired LookupDAO lookupDao;
    @Autowired CityDAO cityDao;
    @GetMapping({" /countries", "/" })
    public String countries(Model model,
        @RequestParam Map<String, Object> params
    ) {
        //logic to fetch country list
        return "countries";
    }
    @GetMapping("/countries/{code}")
    public String countryDetail(@PathVariable String code, Model model) {
        //Logic to Populate the country detail in model
        return "country";
    }
    @GetMapping("/countries/{code}/form")
    public String editCountry(@PathVariable String code,
```

```
Model model) {  
    //Logic to call CountryDAO to update the country  
    return "country-form";  
}  
}
```

The following are a few important things from the previous code:

- **@Controller:** This annotation is used to declare a controller that can return view template names to be able to render the view, as well as returning JSON/XML data in the response body.
- **@ResponseBody:** This annotation when present on the method of the controller indicates that the method is going to return the data in the response body, and hence, Spring will not use the view resolver to resolve the view to be rendered. The **@RestController** annotation by default adds this annotation to all its methods.
- **Model:** This instance is used to pass on the data required for building the view.

In case of the listing of countries, the complete HTML is rendered at the server using the Thymeleaf template engine, so we need to obtain the request parameters, if any are present in the URL, and obtain a filtered and paginated list of the countries. We also need to populate the lookups that is the data for the `<select>` controls, which will be used for filtering the data. Let's look at its implementation as follows:

```
@GetMapping("/{countries}", "/")  
public String countries(Model model,  
    @RequestParam Map<String, Object> params  
) {  
    model.addAttribute("continents", lookupDao.getContinents());  
    model.addAttribute("regions", lookupDao.getRegions());  
    model.addAttribute("countries", countryDao.getCountries(params));  
    model.addAttribute("count", countryDao.getCountriesCount(params));  
    return "countries";  
}
```

The previous code is pretty straightforward. We are making use of the DAO classes to populate the required data into the `Model` instance and then returning the view name, which in this case is `countries`. Similarly, the rest of the method implementation can be found in the `ViewController` controller class.

Defining the view templates

We will be using the Thymeleaf template engine for handling server-side templates. Thymeleaf provides various dialects and conditional blocks for rendering the dynamic content within the static HTML. Let's look at some simple syntactical element of Thymeleaf, as follows:

```
<!-- Dynamic content in HTML tag -->
<div class="alert alert-info">[[${country.name}]]</div>

<!-- Dynamic attributes -->
<span th:class="|alert ${error ? 'alert-danger':
_|}">[[${errorMsg}]]</span>

<!-- Looping -->
<ol>
  <li th:each="c : ${countries}">
    [[${c.name}]]
  </li>
</ol>

<!-- Conditionals -->
<div class="alert alert-warning" th:if="${count == 0}">No results
found</div>

<!-- Custom attributes -->
<div th:attr="data-count=${count}"></div>

<!-- Form element value -->
<input type="text" th:value="${country.name}" name="name" />
```

From the previous examples, we can observe that the items to be evaluated by Thymeleaf are prefixed with `th:` and any content to be rendered between the tags can be done either using `th:text` or `[[${variable}]]`. The latter syntax has been introduced in Thymeleaf 3. This was a very short primer, as going in to depth on Thymeleaf is out of the scope of this book. A beautiful guide explaining different parts of the template can be found at <http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>.

Configuring a Thymeleaf template engine

In order to use the Thymeleaf template engine with Spring MVC, we need to do some configuration wherein we set up the Thymeleaf template engine and update Spring's view resolver to use the template engine to resolve any views. Before moving further, we need to define required dependencies in `pom.xml` as follows:

```
<dependency>
  <groupId>org.thymeleaf</groupId>
  <artifactId>thymeleaf-spring5</artifactId>
  <version>${thymeleaf.version}</version>
</dependency>
<dependency>
  <groupId>nz.net.ultraq.thymeleaf</groupId>
  <artifactId>thymeleaf-layout-dialect</artifactId>
  <version>${thymeleaf-layout-dialect.version}</version>
</dependency>
```

Let's define the configuration view resolver in order, starting with setting up the template resolver as follows:

```
@Bean
public ClassLoaderTemplateResolver templateResolver() {
    ClassLoaderTemplateResolver templateResolver
        = new ClassLoaderTemplateResolver();
    templateResolver.setPrefix("templates/");
    templateResolver.setSuffix(".html");
    templateResolver.setTemplateMode(TemplateMode.HTML);
    templateResolver.setCacheable(false);
    return templateResolver;
}
```

The previous configuration sets the template location that the template engine will use to resolve the template files. Next is to define the template engine, which will make use of `SpringTemplateEngine` and the template resolver defined earlier, as follows:

```
@Bean
public SpringTemplateEngine templateEngine() {
    SpringTemplateEngine templateEngine = new SpringTemplateEngine();
    templateEngine.setTemplateResolver(templateResolver());
    templateEngine.addDialect(new LayoutDialect());
    return templateEngine;
}
```


In the previous configuration, we make use of the Thymeleaf Layout Dialect (<https://github.com/ultraq/thymeleaf-layout-dialect>) created by *Emanuel Rabina*. This layout dialect helps us in creating a view decorator framework wherein all the templates will be decorated with a base template and the decorated templates just provide the necessary content to complete the page. So all the headers, footers, CSS, scripts, and other common HTML can be placed in the base template. This prevents redundancy to a great extent. In our sample app, the `base.html` file present in `worldgdp/src/main/resources/templates` is the base template that is used by other templates.

Next is to define a Thymeleaf view resolver that will override Spring's default view resolver, as follows:

```
@Bean
public ViewResolver viewResolver() {
    ThymeleafViewResolver viewResolver = new ThymeleafViewResolver();
    viewResolver.setTemplateEngine(templateEngine());
    viewResolver.setCharacterEncoding("UTF-8");
    return viewResolver;
}
```

The previous configuration is available in the `com.packt.config.ViewConfiguration` class.

Managing static resources

If you look back at the `com.nilangpatel.worldgdp.AppConfiguration` class, you will see that we have overridden the `addResourceHandlers` method of `WebMvcConfigurer` interface. In the method implementation shown in the following code, we have mapped the static resources prefix URL `/static/**` to the static resources location `/static/` in the `webapp` directory:

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/static/**")
        .addResourceLocations("/static/");
}
```



We have added a few static resources (both CSS and JavaScript) in the `/src/main/webapp/static` folder of the project. Please download the code of this chapter and refer to them side by side.

Creating the base template

We mentioned before that we will be using the Thymeleaf Layout Dialect to create a base template and use the base template to decorate all other templates. The base template will contain all the CSS links, JavaScript source file links, the header, and the footer, as shown in the following code:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout">
  <head>
    <title layout:title-pattern="$CONTENT_TITLE - $LAYOUT_TITLE">World In
Numbers</title>
    <meta name="description" content="" />
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <!-- Include all the CSS links -->
  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-primary">
      <a class="navbar-brand" href="#">WORLD IN NUMBERS</a>
      <div class="collapse navbar-collapse" id="navbarColor01">
        <ul class="navbar-nav mr-auto">
          <li class="nav-item active">
            <a class="nav-link" th:href="@{/countries}">Countries</a>
          </li>
        </ul>
      </div>
    </nav>
    <div class="container">
      <div class="content">
        <div layout:fragment="page_content">
          <!-- Placeholder for content -->
        </div>
      </div>
    </div>
    <div class="modal" id="worldModal" >
    </div>
    <footer id="footer"></footer>
    <!-- /.container -->
    <!-- Include all the Javascript source files -->
    <th:block layout:fragment="scripts">
      <!-- Placeholder for page related javascript -->
    </th:block>
  </body>
</html>
```

The two main important parts of the following template are as follows:

- `<div layout:fragment="page_content"></div>`: The other templates that use the base template as decorator provide their HTML within this section. Thymeleaf Layout Dialect at runtime decorates this HTML with the content from the base template.
- `<th:block layout:fragment="scripts"></th:block>`: Similar to the HTML previous content, any page-specific JavaScript or links to any specific JavaScript source files can be added within this section. This helps in isolating page-specific JavaScript in their own pages.

Any template that wants to use the base template as the decorator will declare this attribute, `layout:decorate="~{base}"`, in the `<html>` tag. We will not go into the content of individual templates as it's mostly HTML. All the templates can be found at the location `worldgdp/src/main/resources/templates`. We have three templates:

- `countries.html`: This is for showing the countries' list with filtering and pagination
- `country-form.html`: This is for editing a country's detail
- `country.html`: This is for showing a country's detail

Logging configuration

Before we jump into the rest of the steps to develop an application, it is good practice to define a log level and format. It is, however, optional but good practice to print the logs in a desired format, along with various logging levels. For this, add an XML file called `logback.xml` with following content in it:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <Pattern>
        %d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
      </Pattern>
    </layout>
  </appender>
  <logger name="com.nilangpatel.worldgdp" level="debug" additivity="false">
    <appender-ref ref="STDOUT" />
  </logger>
  <root level="debug">
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

```
</root>
</configuration>
```

Logback was developed as a successor to the popular Log4j project, and is used as a logging framework for Java applications. This configuration defines the pattern, along with the logging level. To enable logback in your application, you need to add following dependencies to `pom.xml`:

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>${logback.version}</version>
</dependency>
```

Running the application

As we have already configured the deployment to Tomcat, you should have the application running now. You can always download the source code for this book; find the source code under the `worldgdp` folder. After downloading, you have to build it using Maven, as follows:

```
$ mvn package
```

The preceding command will run the tests as well. The WAR file `worldgdp.war`, present in the `target`, can be uploaded to Tomcat through the Manager app or copied to the `TOMCAT_HOME/webapps` folder. Tomcat will then explode the archive and deploy the app.

The following are some of the screenshots of the application in action, starting with the listing page:

WORLD IN NUMBERS

Countries

Countries

Search by name

Q Search

Continent

Region

Asia

Southern and Central Asia

Code	Name	Continent	Region	Area
AFG	Afghanistan	Asia	Southern and Central Asia	652090.0
BGD	Bangladesh	Asia	Southern and Central Asia	143998.0
BTN	Bhutan	Asia	Southern and Central Asia	47000.0
IND	India	Asia	Southern and Central Asia	3287263.0

Next is the page that displays the country details:



The form that is used to edit the country details is shown in the following screenshot:

Countries / India / Editing India(IND / IN)

Editing India(IND / IN)

Name	Local Name	Capital
<input type="text" value="India"/>	<input type="text" value="Bharat/India"/>	<input type="text" value="New Delhi"/>
Continent	Region	
<input type="text" value="Asia"/>	<input type="text" value="Southern and Central Asia"/>	
Head Of State	Government	
<input type="text" value="Kocheril Raman Narayanan"/>	<input type="text" value="Federal Republic"/>	
Independence	Surface Area	
<input type="text" value="1947"/>	<input type="text" value="3287263.0"/>	
Population	Life Expectancy	
<input type="text" value="1013662000"/>	<input type="text" value="62.5"/>	

Then we have popups that are used to add a new city to the country, as shown in the following screenshot:

City

Name
<input type="text"/>
District
<input type="text"/>
Population
<input type="text"/>