Mastering Spring Application Development

Spring is an open source Java application development framework to build and deploy systems and applications running on the JVM. It is the industry standard for Web development and the most popular framework among Java developers. It makes it easy to build modular and testable Web applications by using the Model-View-Controller paradigm and dependency injection.

Mastering Spring Application Development will take you on a journey from developing simple applications with Spring Data and MongoDB to caching your application with Spring Cache. You will discover how Thymeleaf will help you develop applications faster and how to configure it, and how to master dependency. Finally, you will get to grips with Web services by creating RESTful services and SOAP services using Spring Web Service framework, helping you to write Web service client code. You will learn how to set up a spring cache and tie your caching code with business logic.

Who this book is written for
If you are a Java developer with experience in developing applications with Spring, then this book is perfect for you. A good working knowledge of Spring programming conventions and applying dependency injections is recommended to make the most of this book.

What you will learn from this book
- The best approach to configuring Spring applications
- Read as well as manipulate data in MongoDB using Spring Data
- Configure Spring Batch to schedule jobs
- Use Spring Hbase template implementation
- Bootstrap your application with Spring Boot
- Create and use Apache Hadoop jobs to handle big data
- Integrate Spring with Thymeleaf
- Create RESTful services and SOAP services using Spring Web Service framework
- Develop maven applications by using the spring-integration-ftp package
- Configure Hadoop jobs in the Spring framework

Who this book is written for
If you are a Java developer with experience in developing applications with Spring, then this book is perfect for you. A good working knowledge of Spring programming conventions and applying dependency injections is recommended to make the most of this book.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Spring Mongo Integration'
- A synopsis of the book’s content
- More information on Mastering Spring Application Development

About the Author

**Anjana Mankale** is a tech lead and has 8 years of experience in developing web applications. She has developed applications for healthcare, e-commerce portals, media portals, and content management systems using Spring and Struts2. She is extensively involved in application design and implementation. She has worked on Amazon Cloud and Spring Web Services and has recently been involved in deploying and designing cloud-based multitenant applications. Anjana has also authored a cookbook, *Spring Security 3.x Cookbook*, Packt Publishing.

Anjana is passionate about blogging (http://jtechspace.blogspot.in/), where she shares her write-ups and technical code that she has worked on.
Mastering Spring Application Development

Spring is an open source Java application development framework that is used to build and deploy systems and applications that run on a JVM. It makes efficiently built modular and testable web applications, by using a Model-View-Controller paradigm and dependency injection. It seamlessly integrates with numerous frameworks (such as Hibernate, MyBatis, Jersey, and so on), and reduces boilerplate code when using standard technologies, such as JDBC, JPA, and JMS.

The purpose of this book is to teach intermediate-level Spring developers to master Java application development with Spring, applying advanced concepts and using additional modules to extend the core framework. This is done to develop more advanced, strongly integrated applications.

What This Book Covers

Chapter 1, Spring Mongo Integration, demonstrates the integration of a Spring MVC with MongoDB along with installing MongoDB, to create database and collections.

Chapter 2, Messaging with Spring JMS, teaches you to install Apache ActiveMQ and different types of messaging. This chapter also demonstrates the creation of multiple queues and communicating with these queue using Spring templates with the help of screenshots.

Chapter 3, Mailing with Spring Mail, creates a mailing service and configures it using the Spring API, and demonstrates how to send mails with attachments using MIME messages.

Chapter 4, Jobs with Spring Batch, illustrates how Spring Batch can be used to read an XML file, and also how to create Spring-based batch applications to read a CSV file. This chapter also demonstrates how to write simple test cases using Spring Batch.

Chapter 5, Spring Integration with FTP, gives you an overview of different types of adapters, such as inbound and outbound adapters, with an outbound gateway and its configurations. This chapter also looks into two important classes, FTPSessionFactory and FTPsSessionFactory, by using getter and setter.

Chapter 6, Spring Integration with HTTP, takes you through the use of a multivalue map to populate a request and put the map in the HTTP header. Also, it will provide you with information about HTTP and Spring integration support, which can be used to access HTTP methods and requests.
Chapter 7, *Spring with Hadoop*, shows how Spring integrates with Apache Hadoop and provides Map and Reduce processes to search and count data. The chapter also discussed installing a Hadoop instance on Unix machines and configuring Hadoop jobs in a Spring framework.

Chapter 8, *Spring with OSGI*, develops a simple OSGI application, and also demonstrates how a Spring dynamic module supports OSGI development and reduces the creation of files, thereby making things easier with configuration.

Chapter 9, *Bootstrap your Application with Spring Boot*, starts with setting up a simple Spring boot project, along with the process of using a Spring Boot to bootstrap applications. This chapter also gives information about how a Spring Boot supports a cloud foundry server and helps to deploy applications on cloud.

Chapter 10, *Spring Cache*, implements our own caching algorithm and teaches you to make a generic algorithm. This chapter also discusses the classes and interface that support a caching mechanism in a Spring Framework.

Chapter 11, *Spring with Thymeleaf Integration*, integrates the Thymeleaf templating engine into a Spring MVC application, and also uses a Spring Boot to start Spring with a Thymeleaf application.

Chapter 12, *Spring with Web Service Integration*, integrates JAX_WS with Spring Web Service. It demonstrates how to create spring Web services and an endpoint class, accessing the web service by accessing the WSDL URL.
Spring Mongo Integration

MongoDB is a popular NoSQL database and is a document-based one too. It is written using the popular and powerful C++ language, which makes it a document-oriented database. Queries are also document-based, and it also provides indexing using JSON style to store and retrieve data. MongoDB works on the concept of collection and documentation.

Let's look at few terminology differences between MySQL and MongoDB:

<table>
<thead>
<tr>
<th>MySQL</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>Collection</td>
</tr>
<tr>
<td>Row</td>
<td>Document</td>
</tr>
<tr>
<td>Column</td>
<td>Field</td>
</tr>
<tr>
<td>Joins</td>
<td>Embedded documents linking</td>
</tr>
</tbody>
</table>

In MongoDB, a collection is a set or a group of documents. It is the same as RDBMS tables.

In this chapter, we shall start by setting up a MongoDB NoSQL database and will integrate a spring application with MongoDB to perform CRUD operations. The first example demonstrates updating single document values. The second example considers an order use case where it requires two document references to be stored in the collection. It demonstrates the flexibility in referencing different documents of MongoDB using objectId references.

We need to go for a NoSQL database only if the applications have heavy write operations. MongoDB also suits the cloud environment very well, where we can take copies of databases easily.
In the next section, we shall see how we can get started with MongoDB, beginning with installing it, using the Spring Framework, and integrating MongoDB. To get started, we shall show basic Create, Retrieve, Update, and Delete (CRUD) operations with various use cases.

## Installing MongoDB and creating a database

In this section we shall install MongoDB and create a database:

2. Configure the data folder by executing the following command in the bin folder:
   ```
   >mongod.exe -dbpath e:\mongodata\db
   ```
4. Execute the following command:
   ```
   >show database
   ```
   The `>show dbs` command also works fine with MongoDB.
5. Execute the following command to create a new database, namely `eshopdb`.
   ```
   >use new-eshopdb
   ```
6. Executing `> show dbs` will still show that `eshopdb` hasn't been created yet; this is because it doesn't contain any collections. Let's add some collections in the next step, once a collection is added.
7. Execute the following snippet in the Command Prompt. The following snippets will insert sample documents into the collection:
   ```
   db.eshopdb.insert({cust_id:1,name:"kishore",address:"jayangar"})
   db.eshopdb.insert({cust_id:2,name:"bapi",address:"HAL Layout"})
   db.eshopdb.insert({cust_id:3,name:"srini",address:"abbigere street"})
   db.eshopdb.insert({cust_id:4,name:"sangamesha",address:"Kattarigupee layout"})
   ```
Setting up a batch file for MongoDB

It’s always easy to create a batch file to start MongoDB, and it’s best to create a script file to start Mongo. This way, we won’t have an error with the configuration. This would also save us a lot of time.

1. Create a `mongodbstart.bat` file.
2. Edit the file and type in the following command and save it:
   
   ```
   cd E:\MONGODB\mongo\bin
   mongod -dbpath e:\mongodata\db
   ```

   The next time you want to start MongoDB, just click on the batch file.

Order use case with Spring and MongoDB

Let us look at the Order use case to implement a simple CRUD operation using Spring and MongoDB. We are performing CRUD operations on Product, Customer, and Order documents. The scenario is this: a customer selects a product and places an order.

Following is the Order use case. The actor is the application user and will have the following options:

- CRUD operation on Product Document
- CRUD operation on Customer Document
- CRUD operation on Order by selecting Product and Customer
- Saving the Product Document Object ID and Customer Document Object ID in Order Document

Mapping a Mongo document to Spring Bean

Spring provides a simple way to map Mongo documents. The following table depicts the mapping of Bean with MongoDB collections:

<table>
<thead>
<tr>
<th>Bean</th>
<th>Mongo Collections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer.java</td>
<td>db.customer.find()</td>
</tr>
<tr>
<td>Order.java</td>
<td>db.order.find()</td>
</tr>
<tr>
<td>Product.java</td>
<td>db.product.find()</td>
</tr>
</tbody>
</table>
Setting up a Spring-MongoDB project

We need to create a simple web application project using Maven.

1. Execute the following command in your Maven command prompt:
   ```
   mvn archetype:generate -DgroupId=com.packtpub.spring -DartifactId=spring-mongo -DarchetypeArtifactId=maven-archetype-webapp
   ```

2. Create a simple Maven project with a web application archetype. Add the latest 4.0.2.RELEASE spring dependency.

3. The following is an extract from the `pom.xml` file. These are the mandatory dependencies to be added to the `pom.xml` file.
   ```xml
   <!-- Spring dependencies -->
   <dependency>
     <groupId>org.mongodb</groupId>
     <artifactId>mongo-java-driver</artifactId>
     <version>2.9.1</version>
   </dependency>
   <dependency>
     <groupId>org.springframework.data</groupId>
     <artifactId>spring-data-mongodb</artifactId>
     <version>1.2.0.RELEASE</version>
   </dependency>
   <dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-context-support</artifactId>
   </dependency>
   ```
Application design

The following table contains the classes used to develop a simple CRUD application. The request flows from controller to model and back. The Repository classes are marked with the @Repository annotation and connect to MongoDB using the mongoTemplate class.

<table>
<thead>
<tr>
<th>Controller</th>
<th>Model</th>
<th>JSP</th>
<th>Bean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Controller.java</td>
<td>Customer Repository.java</td>
<td>customer.jsp</td>
<td>Customer.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td>editcustomer.jsp</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>allcustomers.jsp</td>
<td></td>
</tr>
<tr>
<td>Order Controller.java</td>
<td>Order Repository.java</td>
<td>order.jsp</td>
<td>Order.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td>editorder.jsp</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>allorders.jsp</td>
<td></td>
</tr>
<tr>
<td>Product Controller.java</td>
<td>Product Repository.java</td>
<td>product.jsp</td>
<td>Product.java</td>
</tr>
<tr>
<td></td>
<td></td>
<td>editproduct.jsp</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>allproducts.jsp</td>
<td></td>
</tr>
</tbody>
</table>
Application implementation of Spring with MongoDB

The following are the steps for the implementation of the Spring4MongoDB_Chapter1 application:

1. Create a web-based Maven project with the name Spring4MongoDB_Chapter1.
2. Import the project into Eclipse for the implementation. I have used Eclipse Juno.

We need to create the controller to map the requests.

The controller request is mapped to the GET and POST methods, as shown in the following table:

<table>
<thead>
<tr>
<th>Request</th>
<th>Request Method</th>
<th>Model Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>/product</td>
<td>GET</td>
<td>productList</td>
</tr>
<tr>
<td>/product/save</td>
<td>POST</td>
<td>productList</td>
</tr>
<tr>
<td>/product/update</td>
<td>POST</td>
<td>productList</td>
</tr>
<tr>
<td>/product/geteditproduct</td>
<td>GET</td>
<td>productAttribute</td>
</tr>
<tr>
<td>/product/deleteproduct</td>
<td>GET</td>
<td>productAttribute</td>
</tr>
<tr>
<td>/product/getallproducts</td>
<td>GET</td>
<td>productList</td>
</tr>
</tbody>
</table>

Following is the implementation of ProductController.java. We have used the @Controller annotation to indicate that the ProductController.java class is a controller class. The @Autowired annotation ties the ProductRepository class with the ProductController.java file.

The property productList is a list of type Product that holds the products that are to be displayed on screen. The @PostConstruct annotation will call the method decorated by it. Once the constructor of the class is called and all properties are set, and before any business methods are called, it's worthy to note as it's only called once.

```java
@Controller
public class ProductController {
    @Autowired
    private ProductRepository repository;
    private List<Product> productList;
    public ProductController() {
        super();
    }
    @PostConstruct
```
public void init(){
    this.productList=respository.getAllObjects();
}
//to get the list of products
@RequestMapping(value="/product", method = RequestMethod.GET)
public String getaddproduct(Model model) {
    model.addAttribute("productList", productList);
    model.addAttribute("productAttribute", new Product());
    return "product";
}
//to save the product
@RequestMapping(value="/product/save", method = RequestMethod.POST)
public String addproduct(@ModelAttribute Product prod,Model model) {
    if(StringUtils.hasText(prod.getProdid())) {
        respository.updateObject(prod);
    } else {
        respository.saveObject(prod);
    }
    this.productList=respository.getAllObjects();
    model.addAttribute("productList", productList);
    return "product";
}
//to update the edited product
@RequestMapping(value="/product/update", method = RequestMethod.POST)
public String updatecustomer(@ModelAttribute Product prod,Model model) {
    respository.updateObject(prod);
    this.productList=respository.getAllObjects();
    model.addAttribute("productList", productList);
    return "product";
}
//to edit a product based on ID
@RequestMapping(value="/product/geteditproduct", method = RequestMethod.GET)
public String geteditproduct(@RequestParam(value = "prodid", required = true) String prodid, Model model) {
    model.addAttribute("productList", productList);
    model.addAttribute("productAttribute", respository.getObject(prodid));
    return "editproduct";
}
The `Product.java` file has an `@Document` annotation and an `@ID` annotation, which is identified as a MongoDB collection that maps the `Product` entity to `product` collection in MongoDB.

```java
@Document
public class Product {
    /* Bean class product with getter and setters */
    @Id
    private String prodid;
    private Double price;
    private String name;
    public Product() {
        super();
    }
    public String getProdid() {
        return prodid;
    }
    public void setProdid(String prod_id) {
        this.prodid = prod_id;
    }
    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
```
Chapter 1

```java
this.price = price;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
}

The ProductRepository.java file has @Repository annotation. This is the persistence layer, and tells spring that this class performs operations on the database. The connection to Mongo is set up in Mongo template.

ProductRepository.java

@Repository
public class ProductRepository {
    @Autowired
    MongoTemplate mongoTemplate;
    public void setMongoTemplate(MongoTemplate mongoTemplate) {
        this.mongoTemplate = mongoTemplate;
    }

    public List<Product> getAllObjects() {
        return mongoTemplate.findAll(Product.class);
    }

    /**
     * Saves a @link Product.
     */
    public void saveObject(Product Product) {
        Product.setProdid(UUID.randomUUID().toString());
        mongoTemplate.insert(Product);
    }

    /**
     * Gets a @link Product for a particular id.
     */
    public Product getObject(String id) {
        return mongoTemplate.findOne(new Query(Criteria.where("_id").is(id)),
                                           Product.class);
    }

    }
```
/**
 * Updates a {@link Product} name for a particular id.
 */
public void updateObject(Product object) {
    Query query = new Query();
    query.addCriteria(Criteria.where("_id")
            .is(object.getProdid()));
    Product prod_tempObj = mongoTemplate.findOne(query,
            Product.class);
    System.out.println("cust_tempObj - " + prod_tempObj);
    //modify and update with save()
    prod_tempObj.setName(object.getName());
    prod_tempObj.setPrice(object.getPrice());
    mongoTemplate.save(prod_tempObj);
}

/**
 * Delete a {@link Product} for a particular id.
 */
public void deleteObject(String id) {
    mongoTemplate.remove(new
            Query(Criteria.where("_id").is(id)),Product.class);
}

/**
 * Create a {@link Product} collection if the collection does not
 * already
 * exists
 */
public void createCollection() {
    if (!mongoTemplate.collectionExists(Product.class)) {
        mongoTemplate.createCollection(Product.class);
    }
}

/**
 * Drops the {@link Product} collection if the collection does
 * already exists
 */
public void dropCollection() {
    if (mongoTemplate.collectionExists(Product.class)) {
        mongoTemplate.dropCollection(Product.class);
    }
}
The .jsp file displays the products available and allows the user to perform CRUD operations on the Product bean. The following screenshot is the output of editing product information using the product ObjectId stored in MongoDB.

![Edit Customer](image)

### Product.jsp file

This file serves as a view layer to the user. This has the product creation form and includes a file that lists all the products stored in MongoDB.

```html
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Register Product</title>
</head>
<body>

<h1>Register Product</h1>
<ul>
</ul>
```
If all goes well, you should see the following screen, where you can play around with products. The following screenshot is the output of the **Register Product** and list Product functionality using Spring and MongoDB.
The following `dispatcher-servlet.xml` file shows the configuration for component scan and MongoDB template. It also shows the MongoDB database name configuration.

dispatcher-servlet.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mongo="http://www.springframework.org/schema/data/mongo"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/
    spring-mongo-1.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/
    spring-context-4.0.xsd">
    <context:component-scan base-package="com.packt" />

      <!-- Factory bean that creates the Mongo instance -->
      <bean id="mongo"
        class="org.springframework.data.mongodb.core.MongoFactoryBean">
        <property name="host" value="localhost" />
      </bean>
      <mongo:mongo host="127.0.0.1" port="27017" />
      <mongo:db-factory dbname="eshopdb" />

      <bean id="mongoTemplate"
        class="org.springframework.data.mongodb.core.MongoTemplate">
        <constructor-arg name="mongoDbFactory" ref="mongoDbFactory" />
      </bean>

      <!-- Use this post processor to translate any MongoExceptions thrown in @Repository annotated classes -->
      <bean class="org.springframework.dao.annotation.
            PersistenceExceptionTranslationPostProcessor" />
      <bean id="jspViewResolver"
        class="org.springframework.web.servlet.view.
        InternalResourceViewResolver" />
</beans>
```
You can see that the mongoDbFactory bean has been configured with MongoDB database details. You will also observe that mongoTemplate has also been configured. The property of the mongoTemplate bean is mongoDbFactory bean, and so when the template is called the connection gets established.

Just run the following commands in the MongoDB database in order to test the Order use case:

- `db.order.find()`
- `db.order.remove()`

[RoboMongo is a free tool like Toad to access the MongoDB database.]

**Order management use case**

Let's consider a complex scenario for this section. In the use case that we have considered, the Order use case has customer and product objects in the class. When a user places an order, the user will select a product and customer.

Our aim here is to store the customer and product classes directly in the Order collection in MongoDB. Let's first implement the `Order` class with getter and setters.

**Order.java**

```java
package com.packt.bean;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document
public class Order {
    private String order_id;
    private Customer customer;
    private Product product;
    private String date;
    private String order_status;
    private int quantity;
```
public Order() {
    super();
    // TODO Auto-generated constructor stub
}

@Id
public String getOrder_id() {
    return order_id;
}
public void setOrder_id(String order_id) {
    this.order_id = order_id;
}

public String getDate() {
    return date;
}
public void setDate(String date) {
    this.date = date;
}
public int getQuantity() {
    return quantity;
}
public void setQuantity(int quantity) {
    this.quantity = quantity;
}
public String getOrder_status() {
    return order_status;
}
public void setOrder_status(String order_status) {
    this.order_status = order_status;
}

public Customer getCustomer() {
    return customer;
}
public void setCustomer(Customer customer) {
    this.customer = customer;
}
public Product getProduct() {
    return product;
}
public void setProduct(Product product) {
    this.product = product;
}
Spring Mongo Integration

The next step would be to define the methods in the `OrderRepository.java` file.

Below are the code snippets of the update and save methods in the repository class.

**Creating and inserting Order**

We see that the update `Order` method accepts the `Order` object. We used the `addCriteria()` method to get a particular order based on the object ID. The `Order` object retrieved is stored in the `temp` object. The values are then set to the `temp` object based on the object that is passed to the method. Then, the `mongoTemplate.save(Object)` method is called to update the saved object.

```java
public void updateObject(Order order) {
    Query query = new Query();
    query.addCriteria(Criteria.where("_id").is(order.getOrder_id()));
    Order order_tempObj = mongoTemplate.findOne(query, Order.class);
    order_tempObj.setCustomer(order.getCustomer());
    order_tempObj.setProduct(order.getProduct());
    order_tempObj.setQuantity(order.getQuantity());
    mongoTemplate.save(order_tempObj);
}
```

The `saveObject` method only accepts the `Order` object and sets the ID to the `Order` object before saving it.
We have seen how to perform an update and an insert. The following method is invoked to save the Order details. This shows that `mongoTemplate` has the methods `insert()` and `save()`.

```java
public void saveObject(Order Order) {
    Order.setOrder_id(UUID.randomUUID().toString());
    mongoTemplate.insert(Order);
}
```

Controller to handle requests

The controller class has the customer repository and product repository references as per the use case. The application user needs to select the customer and product to place an order.

The initial Skelton of `OrderController` is shown here:

```java
@Controller
public class OrderController {
    @Autowired
    private OrderRepository repository;
    @Autowired
    private CustomerRepository customerRepository;
    @Autowired
    private ProductRepository productRepository;
    private List<Order> orderList;
    private List<Customer> customerList;
    private List<Product> productList;

    public OrderController() {
        super();
    }
}
```

Adding the `@ModelAttribute` annotation at the Method level

The controller class is to handle the Order requests. The `@ModelAttribute` annotation is added to the method. The product list and customer list is always available as a model attribute to the controller. The following is the code snippet of the `OrderController` class:

```java
@ModelAttribute("orderList")
public List<Order> populateOrderList() {
```

---

[17]
CRUD operations of the OrderController class

The methods are mapped to a particular request, @ModelAttribute("Order"), to make the order object easily accessible at the JSP level. You can observe that using @ModelAttribute at the method level; this will minimize adding @ModelAttribute to the method.

```java
@ModelAttribute("productList")
public List<Product> populateProductList() {
    this.productList = productRespository.getAllObjects();
    return this.productList;
}

@ModelAttribute("customerList")
public List<Customer> populateCustomerList() {
    this.customerList = customerRespository.getAllObjects();
    return this.customerList;
}
```

```java
@RequestMapping(value = "/order", method = RequestMethod.GET)
// request show add order page
public String addOrder(@ModelAttribute("Order") Order order, 
    Map<String, Object> model) {
    model.put("customerList", customerList);
    model.put("productList", productList);
    return "order";
}
```

```java
@RequestMapping(value = "/order/save", method = RequestMethod.POST)
// request to insert the record
public String addorder(@ModelAttribute("Order") Order order, 
    Map<String, Object> model) {
    order.setCustomer(customerRespository.getObject
        (order.getCustomer().getCust_id()));
    order.setProduct(product_respository.getObject
        (order.getProduct().getProdid()));
    respository.saveObject(order);
    model.put("customerList", customerList);
    model.put("productList", productList);
    return "order";
}
```
@RequestMapping(value = "/order/update", method = RequestMethod.POST)
public String updatecustomer(@ModelAttribute("Order") Order order,
    Map<String, Object> model) {
    order.setCustomer(customerRespository.getObject
        (order.getCust_id()));
    order.setProduct(product_respository.getObject
        (order.getProdid()));
    respository.updateObject(order);
    model.put("customerList", customerList);
    model.put("productList", productList);
    return "order";
}

@RequestMapping(value = "/order/geteditorder", method = RequestMethod.GET)
public String editOrder(
    @RequestParam(value = "order_id", required = true)
    String order_id, @ModelAttribute("Order") Order order,
    Map<String, Object> model) {
    model.put("customerList", customerList);
    model.put("productList", productList);
    model.put("Order", respository.getObject(order_id));
    return "editorder";
}

@RequestMapping(value = "/order/deleteorder", method = RequestMethod.GET)
public String deleteorder(
    @RequestParam(value = "order_id", required = true)
    String order_id, @ModelAttribute("Order") Order order,
    Map<String, Object> model) {
    respository.deleteObject(order_id);
    model.put("customerList", customerList);
    model.put("productList", productList);
    return "order";
}

### JSP files

The Order.jsp file demonstrates the use of @ModelAttribute, which gets mapped to the Model Order defined in the controller class. The setter methods set the values to the objects, which minimizes the coding. This showcases a feature in spring, which simplifies the coding process.
Orders.jsp

<h1>Orders</h1>

<ul>
<li><a href="/Spring4MongoDB_Chapter1/customer">Customer</a></li>
<li><a href="/Spring4MongoDB_Chapter1/product">Product</a></li>
</ul>

<form:form action="/Spring4MongoDB_Chapter1/order/save" modelAttribute="Order">
<table>
<tr>
<td>Add your Order:</td>
<td><form:input path="quantity" size="3"/></td>
</tr>
<tr>
<td>Select Product:</td>
<td>
<form:select path="product.prodid">
<form:option value="" label="--Please Select"/>
<form:options items="${productList}" itemValue="prodid" itemLabel="name"/>
</form:select>
</td>
</tr>
<tr>
<td>Select Customer:</td>
<td>
<form:select path="customer.cust_id">
<form:option value="" label="--Please Select"/>
<form:options items="${customerList}" itemValue="cust_id" itemLabel="name"/>
</form:select>
</td>
</tr>
<tr>
<td colspan="2" align="center">
<input type="submit" value="Submit" />
</td>
</tr>
</table>
</form:form>

<%@ include file="allorders.jsp" %>
</body>
</html>
The `allorders.jsp` file displays the list of orders with an option to edit. Use of MongoDB has made displaying the orderList simpler.

### Allorders.jsp

```html
<h1> E-shop Orders</h1>
<table style="border: 1px solid; width: 500px; text-align:center">
<thead style="background:#fffcc">
<tr>
<th>Order Id</th>
<th>Customer Name</th>
<th>Customer Address</th>
<th>Product Name</th>
<th>Product Price</th>
<th>Product Quantity</th>
<th colspan="2"></th>
</tr>
</thead>
<tbody>
<c:forEach items="${orderList}" var="order">
<c:url var="editUrl" value="/order/geteditorder?order_id=${order.order_id}" />
<c:url var="deleteUrl" value="/order/deleteorder?order_id=${order.order_id}" />
<c:url var="addUrl" value="/order/" />
<tr>
<td><c:out value="${order.order_id}" /></td>
<td><c:out value="${order.customer.name}" /></td>
<td><c:out value="${order.customer.address}" /></td>
<td><c:out value="${order.product.name}" /></td>
<td><c:out value="${order.product.price}" /></td>
<td><c:out value="${order.quantity}" /></td>
<td><a href="${editUrl}">Edit</a></td>
<td><a href="${deleteUrl}">Delete</a></td>
<td><a href="${addUrl}">Add</a></td>
</tr>
</c:forEach>
</tbody>
```

The following is a screenshot of the page to add your order:

### Orders

- **Customer**
- **rProduct**

Add your Order: 8
Select Product: Samsung mobile
Select Customer: Ravindra

### E-shop Orders

<table>
<thead>
<tr>
<th>Order Id</th>
<th>Customer Name</th>
<th>Customer Address</th>
<th>Product Name</th>
<th>Price</th>
<th>Quantity</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>7a616788-bb47-4166-bb73-da1f125d6f53</td>
<td>Ravindra</td>
<td>MILKCOLONY</td>
<td>Samsung mobile</td>
<td>23223.0</td>
<td>8</td>
<td>Edit Delete Add</td>
</tr>
<tr>
<td>b6c40f70-653b-41a7-a25e-65fbb43cfc2a</td>
<td>anju</td>
<td>lnagar</td>
<td>Samsung mobile</td>
<td>23223.0</td>
<td>8</td>
<td>Edit Delete Add</td>
</tr>
<tr>
<td>cf6b5bce-cb5e-4a61-8b9e-ec8230523bad</td>
<td>Ravindra</td>
<td>MILKCOLONY</td>
<td>Samsung mobile</td>
<td>23223.0</td>
<td>34</td>
<td>Edit Delete Add</td>
</tr>
</tbody>
</table>
The following is a screenshot of the page to edit your order:

### Orders

- **Customer**
- **rProduct**

Order id: 7a018788-bb47-4168-bb

Quantity: 8

Select Product: samsung mobile

Select Customer: Ravindrav

Submit

### E-shop Orders

<table>
<thead>
<tr>
<th>Order Id</th>
<th>Customer Name</th>
<th>Customer Address</th>
<th>Product Address</th>
<th>Product Name</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>7a018788-bb47-4166-bb73-da1f125d6f53</td>
<td>Ravindrav MILKCOLONY</td>
<td></td>
<td></td>
<td>samsung mobile</td>
<td>23223.0</td>
<td>8</td>
</tr>
<tr>
<td>b6c40f70-653b-41a7-a25e-65fbb43cfc2a</td>
<td>anju</td>
<td>hnagar</td>
<td></td>
<td>samsung mobile</td>
<td>23223.0</td>
<td>8</td>
</tr>
<tr>
<td>c68b5be6-cb5a-4a61-889e-ec8230523bad</td>
<td>Ravindrav MILKCOLONY</td>
<td></td>
<td></td>
<td>samsung mobile</td>
<td>23223.0</td>
<td>34</td>
</tr>
<tr>
<td>c9b9ab9c-f06f-43f3-8555-d17339596d23</td>
<td>Ravindrav MILKCOLONY</td>
<td></td>
<td></td>
<td>samsung mobile</td>
<td>23223.0</td>
<td>8</td>
</tr>
</tbody>
</table>
Summary
In this chapter, we learned how to install MongoDB and create a database and collections. In MongoDB, we have used the latest version of spring that was available during the writing of this chapter. We also learned how to integrate Spring MVC with MongoDB. We have built a CRUD operation. We have also seen the usage of annotations such as @Repository, @Document, and @Controller. In the next chapter, let us see how we can integrate spring message brokers using jms templates.
Where to buy this book

You can buy Mastering Spring Application Development from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.