Spring, Hibernate, Data Modeling, REST and TDD

AGILE JAVA

DESIGN AND DEVELOPMENT

Amritendu De

Contents

Dedication
Copyright
Contributors
Preface
Part I. An Introduction to Data-Driven Development
Chapter 1. Architecture
Chapter 2. Managing a Standalone Entity
Part II. Managing One-to-One Relationships
Chapter 3. One-to-One Unidirectional Relationship
Chapter 4. One-to-One Bi-directional Relationship
Chapter 5. One-to-One Self-Referencing Relationship
Part III. Managing One-to-Many Relationships
Chapter 6. One-to-Many Unidirectional Relationship
Chapter 7. One-to-Many Bi-directional Relationship
Chapter 8. One-to-Many Self-Referencing Relationship
Part IV. Managing Many-to-Many Relationships
Chapter 9. Many-to-Many Unidirectional Relationship
Chapter 10. Many-to-Many Bi-directional Relationship
Chapter 11. Many-to-Many Bi-directional with Join-Attribute Relationship
Chapter 12. Many-to-Many Self-Referencing Relationship
Chapter 13. Many-to-Many Self-Referencing with Join-Attribute Relationship
Part V. Managing Inheritance Relationships
Chapter 14. Single Table Inheritance
Chapter 15. Concrete Table Inheritance
Chapter 16. Class Table Inheritance
Appendix: Setting Up the Workspace

Dedicated to my father, the late Tapan Kumar De

Copyright © 2014 by Amritendu De All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the written prior permission of the author.
Editor - Cathy Reed (www.cathyreedediting.com) Cover design – Iryna Spica (www.spicabookdesign.com)

About the author 

After completing his computer science, engineering degree, Amritendu De has spent the last 12 years working in the IT industry. He has worked for organizations such as HP, IBM and Oracle in various capacities and is currently working with a leading multinational company as a Senior Software Architect. He has worked extensively on Java and related technologies; has been involved in end-to-end product development for CRM, Utilities and Data Center Infrastructure Management business domains; and holds Sun Certified Enterprise Architect (SCEA) certification. He is the author of the Oracle Certified Master Java EE Enterprise Architect Practice Guide and the Oracle Certified Associate Java SE 7 and SE 6 Practice Exams. In his free time, Amritendu likes to watch movies and read detective stories.

About the technical editor

Michael Rocha is an Enterprise Architect and Consultant with more than 25 years of experience. He started his career in IT in the era of Unix based systems and MS-DOS PCs and has been at the forefront of disruptive technologies, including the client-server era, web-based computing, the dot com boom, and most recently in the Cloud, Analytics, Mobile and Social computing. He has extensive experience in Java and JEE and has architected solutions for clients worldwide.

About the code contributors

Lalit Narayan Mishra is an Electronics engineer who has spent the last eight years working on Java and related technologies. He is also a well-known photographer who likes to capture nature when not working on his computer.

Hazekul Alam graduated from West Bengal University of Technology in 2006 with a degree in Computer Engineering. He has worked for organizations like Sears Holding and IBM, has extensive experience with Java and related technologies, and is currently working with Synechron Technologies as Sr. Associate. In his spare time, he likes to watch cricket.

Preface

I have been working on Java based, data driven applications for more than a decade now. I have seen the rise of popular object relational model frameworks solving the object relational impedance mismatch which is a set of technical and conceptual problems encountered in an object oriented program when using a relational database management system. This book takes into consideration all the relationships of data modeling and solves the problem by designing, developing and unit testing using Spring and Hibernate technology. Developers who are working with relational database driven Spring and Hibernate applications will find it very useful. The book will also be helpful for developers who are using Spring for designing business service and data access tiers or service based development which supports front ends like Android, iPhone or mobile browser.

Chapter 1 starts with a discussion of the most popular data driven Java based architecture – Spring and Hibernate. The architecture along with each architected and important tier is discussed in detail. The chapter also covers REST architectural style in detail.

Chapter 2 starts with a simple example of managing a standalone entity. The chapter contains detailed explanations with code examples.

Chapters 3, 4 and 5 focus on one-to-one relationships with a detailed explanation of how to manage a one-to-one unidirectional, bidirectional and self-referencing relationship.

Chapters 6, 7 and 8 focus on one-to-many relationships. If you wish to manage a one-to-many relationship, whether it is a unidirectional, bidirectional or self-referencing relationship, this chapter shows you how.

Chapters 9, 10, 11, 12 and 13 are about managing many-to-many relationships. If you wish to manage a many-to-many relationship of different varieties, these chapters will show you how.

Chapters 14, 15 and 16 are about managing inheritance relationships. The different forms, like Single table, Concrete table and Class table inheritance relationships, are covered in detail.

All the chapters begin with a mock test service tier which can be used for developing the user interface. The user interface team and the service development team can work in parallel, saving crucial dependency time in an agile manner. You will find a detailed explanation of how to achieve this by reading the second chapter.

The chapters contain fragments of the entire source code and therefore I thought it would be useful to provide the entire source code to the readers. I strongly urge that the reader gets hands-on and implements all the code described in this book. Experienced architects and developers may skip as per their judgment. The Github location for the entire application covered in the book is: https://github.com/Spring-Hibernate-Book/spring-hibernate-datamodeling-tdd-rest.The download does not require a password.

The **Appendix** contains the software installations used to develop the application. At the time of publishing the book, the versions may have been upgraded, so only the names are provided. The latest versions available can be downloaded and a small amount of tweaking may be required.

I wish you good luck and happy reading!

Part I. An Introduction to Data-Driven Development

Chapter 1. Architecture

In this chapter, we cover the essentials of forming software architecture, specifically REST based. We will first look at what is software architecture. We will then look at how to select a framework for RESTful Web Services, a web framework, a business-tier technology, and a persistence mechanism. Finally, we will review some popular patterns and best practices and take a look ahead to the application architecture covered in the book. The aim is to set the context for the discussions after this chapter on data driven development using REST architectural style.

What is Software Architecture?

Software architecture denotes the higher-level building blocks of a software system. Like architecture in the real world, software architecture defines the high-level system structure of the software components, the discipline of creating the component hierarchy, and documenting this structure. It also forms a set of architectural design decisions that lead to the structure and the rationale behind it. During the start of the product development, an architect creates a software architecture, which forms the basis of design and development for the technical requirements converted from business requirements.

When we look at how software development started, we mainly see focus on the required functionality and a data-driven flow. However, modern application development has extended the focus on quality attributes such as extensibility, reliability, performance, scalability, mantainability, availability, security, backward compatibility and usability. These quality attributes drive the software architecture to a certain extent and are popularly known as non-functional requirements. There are many architectural patterns and styles that have evolved over time with both the functional and non-functional requirements. A software architecture is typically built on a solid foundation using one of these proven architectural mechanisms. A poor architecture may expose risks, such as a system being unstable, being unable to execute immediate or future business requirements, or being difficult to manage in a production environment.

When the architect initially studies the requirements of the new system to be

developed, he should first check to see if the use case can be designed and implemented using an established architecture pattern. Because it is typically very costly in time and money to change an architecture late in the project, this initial phase is very important. One should be very careful when creating the software architecture because it is not only costly, but also difficult to change and it may put your application at risk.

The software architecture should have a number of goals beyond basic functionality. It should be flexible and be able to adapt to new business requirements and challenges. For example, a system under development may have the requirement of doubling the number of registered users within a year. Each module or component of the architecture should have a single responsibility or an aggregation of cohesive functionality. A specific component should not be aware of the details of other components or modules. Duplication should be avoided and specific functionality should be implemented in one and only one related module.

Communication between stakeholders is another key to good architecture. As with software architecture plans, there are proven patterns and best practices that you can rely on for optimal success here. The most popular way to reduce complexity is by introducing the separation of concerns and views which can be addressed by specific stakeholders.

Modern product development does not allow an upfront, big design; it promotes the use of agile methodology to break the design into more manageable pieces. It should also reduce risk by analyzing the system using modeling techniques and visualizations. For example, a system may be first drafted using enterprise modeling tools like Enterprise Architect or Rational Rose. The entire team may work in parallel describing the design and the architecture using those tools. Finally, the architecture developed can be evaluated by proven techniques such as the Software Architecture Analysis method (SAAM). SAAM was the first documented software architecture analysis method, developed in the 1990s to analyze non-functional requirements and flexibility for change. An in-depth discussion of these techniques is beyond the scope of this book.

What is REST?

REST (REpresentation State Transfer) is an architectural style of networked

systems, such as Web applications. REST was first introduced in 2000 in a Ph.D dissertation by Roy Fielding, who was one of the writers of the HTTP specification. REST is a simple stateless architecture that generally runs over HTTP. A good REST principle to start with is that the interaction between client and server between requests is stateless in nature. Each request should be able to communicate all the information required to process the request. On the server side, all requests are identified by a Universal Resource Identifier (URI). The URI should be unique to a web application. Standard HTTP methods such as POST, GET, DELETE and PUT can be used. The REST architecture scales well to a large number of client requests because it is lightweight in nature and simplifies interaction between client and server. We will also look at some of the REST concepts like resource, representation, connectors and components. A resource is accessible using a global identifier when implementing REST over HTTP. A representation is a document representing the current standing of a resource. A component is a piece of software instruction, the state of which is modifiable via the exposed interfaces. A connector mediates communication among components.

Frameworks for RESTful Web Services

There are three popular RESTful Web Services frameworks: Jersey, Restlet, and Spring. Jersey RESTful Web Services is an open-source framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs, constituted by Oracle's Java EE 7 JAX-RS specifications (JSR 311 & JSR 339). Jersey is a Reference Implementation. Restlet is a lightweight, open-source REST framework for the Java platform. Restlet API derived the name of its classes directly from the REST concepts (resource, representation, connector, component, media type, language), indicating that one is able to think RESTfully and to more easily translate a solution into code. The third framework for building RESTful Web Services is the Spring framework where REST support is assimilated flawlessly into Spring MVC. In this book, we will focus extensively on using the Spring framework to build RESTful services.

Web Frameworks

There are currently two major approaches for doing web development: server-side MVC and client-side MVC. Server-side MVC renders all pages on the

server. Popular examples of this framework include Java Server Faces (JSF), Oracle ADF, JBoss Seam and Spring MVC. Client-side MVC combines Java Server Pages (JSP) and jQuery along with a REST API. There are specialized forms of jQuery like Angular and Backbone that can be used, and a REST API can be developed using Spring REST support. In this book, we will use client-side MVC extensively. We will use JSP and jQuery for building the user interface and Spring MVC REST for the presentation tier. The essential difference between client side MVC and server side MVC is that the client bears most of the logic in client side MVC (extensively using Javascript) whereas in server side MVC the server has to process most of the logic.

Business-Tier Technology

Oracle Java EE 7 talks about two main technology choices for implementing business tiers in a JEE web application. The first approach is to use session beans, which are part of the Enterprise Java Beans 3.2 (JSR 345). The second approach is to use Java API for XML based Web Services (JAX-WS 2.2). The third approach is the non-JEE approach provided by the Spring framework. The Spring framework provides an annotation called @Service which indicates the class is a business service façade implementation. In this book, we will see this annotation widely used in developing the business tier.

Persistence Mechanism

Persistence is the ability to store the state of an object in a non-volatile storage such as hard disk. Oracle Java EE 7 mainly provides two basic mechanisms for implementing the persistence tier of a web application. The first approach is the use of Java Database Connectivity API (JDBC) which provides methods for querying and updating data in a database. The second approach is the use of the Java Persistence API (JPA) which describes the management of relational data in applications. Here, we will use the second approach with Hibernate, an implementation of the JPA specification. Hibernate-specific annotations have been intentionally avoided and coverage restricted to the use of only JPA annotations.

Native App vs. Mobile Browser

One major design decision was the use of client side MVC. An important reason for adopting this architecture to support mobile applications was due to the fact that there is an increasing demand for architectures to be mobile ready. As of this writing, mobile application development should support mainly Android and iPhone devices. I strongly believe in the development of native applications for Android and iPhone because of performance, better usability and better accessibility; while other devices like Windows and Blackberry can make use of mobile browser-based HTML5 and CSS3 apps. Restricting development to a mobile browser or developing native applications for both Android and iPhone devices is an important decision; but .in the end, a native application always gives greater user satisfaction and support.

The Architecture Used in the Book

In this chapter, we have dealt with the most common architecture for Java-based web projects – Spring and Hibernate based. From a Spring point of view, we will look at Spring Web MVC, Spring Core and Hibernate for persistence.

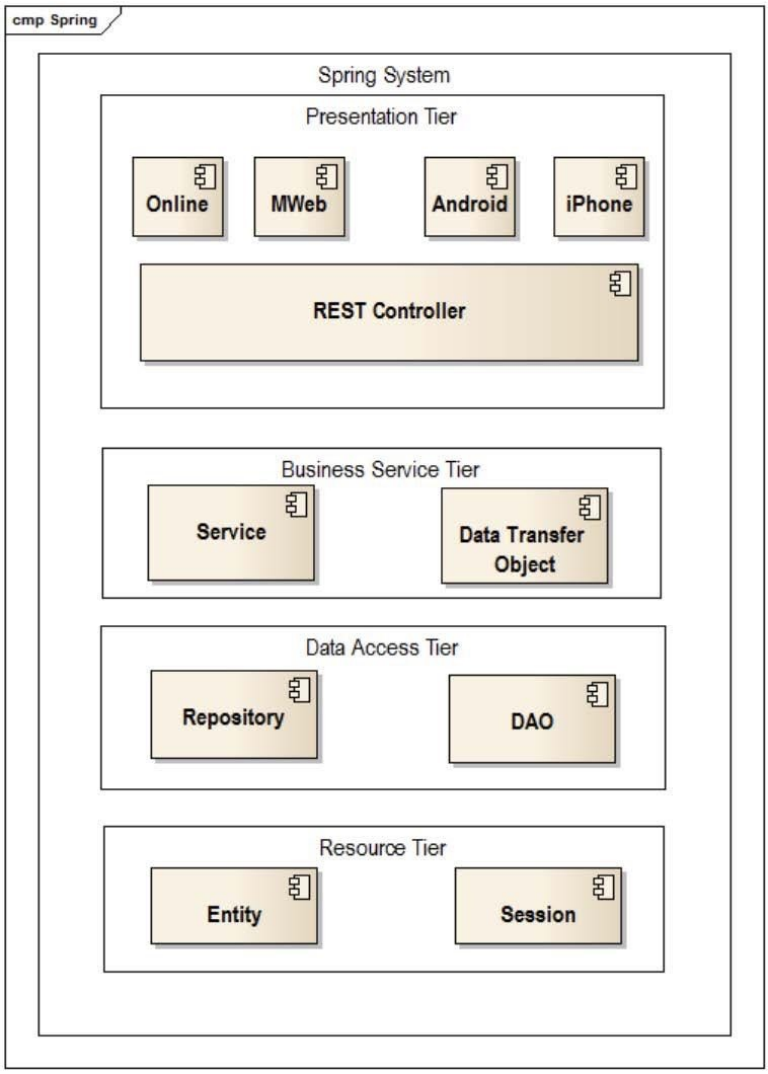

Figure 1-1: Architecture

Figure 1-1 explains the architecture followed in this book. I will attempt to explain the architecture from a bottom-up point of view. The entities are managed using Hibernate implementation. The data access tier is managed by Spring which is also transactional. The business service tier, also implemented by Spring, calls for one or more instances of data access tier to manage the subsystem. The presentation tier contains JSON REST based Spring Controllers to expose the services. The user interface is Java Server Pages (JSP) and jQuery-based for the online system. Other popular user interfaces like Android, iPhone, and mobile browsers are not covered in this book but may consume the same service for development. The idea behind the book is to present the pattern of management of relationships and not to go deep into non-functional requirements like security.

Popular Patterns and Best Practices

There are a number of selected patterns and best practices used in this book. The Business Object pattern is used to separate business data and business logic using an object model. The business objects represent the domain model and are void of any business logic. The Data Access Object pattern is used to encapsulate calls to the underlying data source which also manages the connection with the data source to retrieve and store data. The Session Façade pattern is used as a façade to encapsulate the complexity of interactions between the business objects participating in a workflow. Please note that the Session Façade patterns calls one or more Data Access Objects in a single transaction to store and retrieve data. The calls to the Session Façade are done using Data Transfer Objects which is converted to a Business Object via a mapper. The Data Transfer Object is used to carry data across the tier. The REST based Web Service pattern is used to discover the services which multiple user interfaces calls.

Test-driven Development

The book thoroughly covers test-driven development which is a best practice for agile development. In this approach the developer writes a test first, which initially fails because the desired function or improvement does not exist. Then the developer writes the minimum amount of code to pass the test and finally refactors the code to achieve the accepted standards. Please note that the test cases should not be dependent on each other and can be executed in any sequence or order. Also, the test cases should not be written based on the assumption of any state of the system like hard-coded identifiers. One common mistake is to not treat the test code with respect. Your test code should be treated with the same respect as your production code and must work correctly for positive as well as negative cases.

Summary

REST Architecture generates a simple, efficient, scalable, reliable, secure and extensible design of a Java EE application. The RESTful Web Services have emerged as a great alternative to SOAP-based Web Services due to their lightweight nature, simplicity and ability to transfer data over HTTP. JQuery

which implements an AJAX framework fits like a glove with RESTful Web Services implemented using Spring MVC. In the next chapters, we will start with the journey of taking relationships of data modeling as an example and implementing the different tiers of the architecture described above.

Chapter 2. Managing a Standalone Entity

This chapter is an introduction to the entire process of development with a standalone entity. Following the chapter closely will educate you regarding the design of the tiers of the application along with development and unit testing techniques. First, we will look at the development of the mock user interface and then turn to the development of the REST service. We will be taking examples of data modeling scenarios and build all the tiers of the application. In this chapter, we have taken an example of a standalone entity and will be building all the tiers showing how to manage the entity.

Domain Model

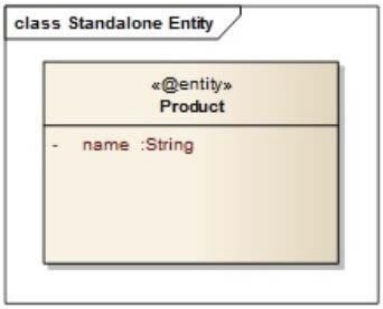

Figure 2-1: A Standalone Entity We will start with a simple standalone entity called Product, shown in Figure 2-1. The properties include name along with an identifier field. The development will be divided into two main broad teams - the user interface team and the service development team.

Develop a User Interface

When the user interface team starts, they do not receive the service ready. Hence they have to create a mock service with an in-memory database. We will divide the user interface development into three major tasks – the data transfer object, the mock service, and the mock user interface. Remember, the mock user interface will be exactly similar to the real user interface with the exception that the database is mock. When the service development team is ready to integrate with the user interface team, we will replace the mock service with the real one.

Develop the data transfer object

The data transfer object acts as a mapper between the user interface and the data access object tier. The internal database design should not be disclosed to the user interface; hence mapping is required with a data transfer object.

```
public class ProductDto implements Serializable {


private static final long serialVersionUID = 1L; private Integer id;
  private String name;

  // getters and setters
  }
```

Develop the mock service

The mock service is a Spring JSON based controller, which saves the data to an in-memory database. Let's start with a simple add to the database. The design of the in-memory database is based on Singleton pattern which is implemented using ENUM as a best practice. The identifier field starts with 1 and is incremented by 1 in successive additions.

```
public enum ProductInMemoryDB{
INSTANCE;
private static List<ProductDto> list = new ArrayList<ProductDto>(); private
static Integer lastId = 0;



public Integer getId() { return ++lastId; }

public void add(ProductDto productDto) {
productDto.setId(getId());
list.add(productDto);

                                                    }

                                                    }
```

Using this in-memory database, we will look at the Spring controller using the @Controller annotation. The @RequestMapping annotation is used to map URLs such as *standalone*mock in this instance, which is applied to all the methods in the class. The method created in the ProductMockController class has a URL *standalone*mock/create accepting a POST request. Also note that the method is annotated with @ResponseBody which means the return character is written to the HTTP body.

```
@Controller
@RequestMapping(value="standalonemock")
public class ProductMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ProductDto product){
  ProductInMemoryDB.INSTANCE.add(product); }

                                                    }
```

Next we will look at the method which is responsible for selection of all the rows. First, we will look at the in-memory database operations which the controller is going to use.

```
public class ProductInMemoryDB {
public List<ProductDto> findAll() { return list; }

public ProductDto findById(Integer id) {
for (ProductDto dto:list)
if (dto.getId()==id) return dto;
return null;

}

}
```

The controller will be using each of these operations. The *findAll returns all instances of ProductDto, and* findById returns a specific row with the given identifier. Both the methods use HTTP GET to retrieve the instances of rows from the in-memory database.

```
public class ProductMockController {
@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody List<ProductDto> findAll(){
return ProductInMemoryDB.INSTANCE.findAll();

}

@RequestMapping(value="/findById/{productid}",
method=RequestMethod.GET) public @ResponseBody ProductDto findById(
@PathVariable("productid") Integer productid){
return ProductInMemoryDB.INSTANCE.findById(productid); }

}
```

Next we will look at the remove operation which removes a specific row with an identifier.

```
public class ProductInMemoryDB {
public void remove(Integer id) {
ProductDto toRemove = null;
for (ProductDto dto:list)
```

```
if (dto.getId()==id) toRemove = dto;
if (toRemove!=null) list.remove(toRemove);

                                                    }

                                                    }
```

The /remove operation deletes a specific row with the given identifier using HTTP POST method.

```
public class ProductMockController {
@RequestMapping(value=”/remove/{productid}”,
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT) public void remove(@PathVariable(“productid”)
Integer productid){
ProductInMemoryDB.INSTANCE.remove(productid); }

                                                    }
```

The edit method updates a specific row with the details. Let’s look at the in-memory database operations which the controller is going to use.

```
public class ProductInMemoryDB {
public void edit(ProductDto productDto) {
for (ProductDto dto:list)
if (dto.getId()==productDto.getId())
dto.setName(productDto.getName());
  }

                                                    }
```

The /edit operation updates a row with details using HTTP POST method.

```
public class ProductMockController {
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
  public void edit(@RequestBody ProductDto product){
ProductInMemoryDB.INSTANCE.edit(product);
  }

                                                    }
```

Develop the mock user interface

We'll first build a mock user interface to demonstrate how we can integrate with the mock service. This is a helpful tool for the user interface team to do their work in parallel with the service development team. Later we will integrate this with the actual service. We will be using JSP for the user interface and JQuery with AJAX to connect with the REST service that was developed. REST services can be utilized as a plain URL as well as possessing a body passing JSON objects. For the retrieval and remove operations, plain URL will be used, whereas for add and update operations, a JSON object will be passed to the REST service. Let's first look at creating the user interface shown in Figure 2-2.

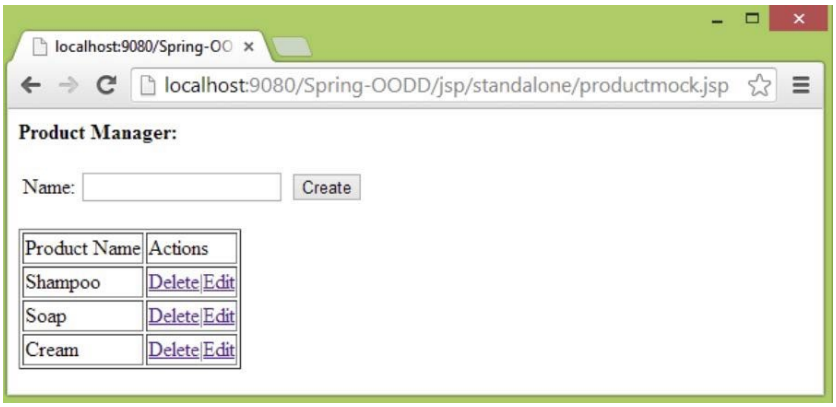

Figure 2-2: A screenshot of the Product manager user interface Let's first look at the JSP fragment which constitutes the body of the page. An input box where the user can enter the product name is being established which is tied to the physical body of the page. Once the page is submitted, the call goes to a JavaScript method which is discussed later. Also note that when the page loads again, a JavaScript method is called which fills the grid.

```
<body onload="loadObjects()">
 <div id="container">
  <div>
    <p><b>Product Manager:</b></p> </div>
  <form method="post" id="productForm"> <table>
      <tbody>
<tr>
  <td>Name:</td>
  <td>
    <input name="productid" id="productid" type="hidden"> <input
```

```
name=”productname” id=”productname” type=”text”> </td>
  <td>
    <input type=submit value=”Create” id=”subButton” onclick=”return
methodCall()”> </td>
      </tr>
     </tbody>
   </table>
  </form>
  <div id=”productFormResponse”></div> </div>
</body>
```

The JavaScript calls are discussed in sequence of calls. First we will see the JavaScript method which is called when the page is loaded. The JQuery Ajax method is used to perform an AJAX request. The url points to the REST mock service we developed earlier, which is an HTTP GET request, and the data type expected in the server response is JSON based. When the function is successful, another JavaScript method is called, and if there is an error a JavaScript alert with the error is given as output.

```
function loadObjects(){
$.ajax({
url : “/Spring-OODD*standalone*mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

                                                {

processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“productname”).value=””; alert(“Error Status Load
Objects:”+textStatus); }

                                                });

return false;

                                                }
```

The JavaScript method which is called when the above service call is successful

is shown below. Please note that the method loops over the response JSON and populates the div tag with the table structure consisting of the name and the identifier. Also, each row in the table consists of links to delete and edit the row in the table.

```
function processResponseData(responsedata){
var dyanamicTableRow=”<table border=1>”+
“<tr>” +
“<td>Product Name</td>”+
“<td>Actions</td>”+
“</tr>”;
var dataRow=””;
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue); });
dyanamicTableRow=dyanamicTableRow+dataRow+”</table>”;
document.getElementById(“productFormResponse”).innerHTML=dyanamicTab
}

function generateTableData(itemvalue){
var dataRow=”<tr>” +
“<td>” +itemvalue.name+”</td>”+
“<td>” +
“<a href=# onclick=deleteObject(“+itemvalue.id+”)>Delete</a>”+
“|<a href=# onclick=editObject(“+itemvalue.id+”)>Edit</a>”+
“</td>”+
“</tr>”;
return dataRow;

}
```

We have witnessed how the page loading works. Now let’s look at the create functionality. Please go back and look at the JSP fragment which calls a JavaScript method when the name is entered and the page is submitted.

```
function methodCall(){
var buttonValue = document.getElementById(“subButton”).value;
if(buttonValue==”Create”){
create();
}else if(buttonValue==”Update”){
update();

}
```

ʃ

return false;

}

First we will look at create. The call to create is an HTTP POST call where the name of the product is passed as a JSON object. Please remember to pass request headers Accept and Content-Type which are marked as application/json.

```
function create(){
var name = $("#productname").val();
var formData={"name":name};
$.ajax({
url : "/Spring-OODD*standalone*mock/create", type: "POST",
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)
```

{

```
document.getElementById("productname").value="";
document.getElementById("subButton").value="Create"; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("productname").value=""; alert("Error Status
Create:"+textStatus);
```

}

});

return false;

}

Next we will look at the edit option. We have seen how the edit link is embedded while creating the table. The edit option also is an HTTP POST call which accepts the headers, and a JSON object containing the name and an

identifier is passed as a parameter.

```
function update(){
var name = $("#productname").val();
var id = +$("#productid").val();
var formData={"id":id,"name":name};
$.ajax({
url : "/Spring-OODDstandalonemock/edit", type: "POST",
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)
{
document.getElementById("productname").value="";
document.getElementById("subButton").value="Create"; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("productname").value=""; alert("Error Status
Update:"+textStatus);
}
});
return false;
}
```

The editObject and the viewObject Javascript method call allow the selection of a record for an edit operation.

```
function editObject(productid){
editurl="/Spring-OODDstandalonemock/findById/"+productid; var
productForm={id:productid};
$.ajax({
url : editurl,
type: "GET",
data : productForm,
dataType: "json",
```

```
success: function(data, textStatus, jqXHR)
                                                    {
viewObject(data);
document.getElementById(“subButton”).value=”Update”;
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Find Object:”+textStatus); }
                                                    });
                                                    }
function viewObject(data){
document.getElementById(“productname”).value=data.name;
document.getElementById(“productid”).value=data.id; }
```

Lastly, we will look at the delete link. The delete link holds the identifier and passes to the JSON mock service as a parameter. The delete call is also an HTTP POST call.

```
function deleteObject(productid){
var productForm={id:productid};
delurl=”/Spring-OODD<i>standalone</i>mock/remove/”+productid; $.ajax({
url : delurl,
type: “POST”,
data : productForm,
dataType: “json”,
success: function(data, textStatus, jqXHR)
                                                    {
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus);
                                                    }
                                                    });
                                                    }
```

This concludes the first part of the maturation process. While the user interface team is busy creating the interface, the service development team would have moved at the same pace. In the next part, let's look at how we will create the actual database driven service.

Develop the Service

The service development will be divided into four major steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST service which will be integrated with the user interface. Let's go one at a time and go bottom up starting with the entity and go up till the REST service.

Develop the resource (entity) tier We will follow the Test Driven Development (TDD) approach for developing the service tier. First we will write a test for each of the methods and then subsequently the method definition will follow. Let's look at the entity test followed by how to create the Product entity. The entity test is run with Spring configuration using @RunWith annotation which loads the Spring ApplicationContext. The ApplicationContext location is context.xml which is part of the classpath. If you want to know more about how to code the ApplicationContext you may look at the Appendix. The test cases are transactional in nature and are rolled back when complete. The Hibernate 4 Session is embedded in the entity test which is configured in the Spring configuration. The entity test checks for basic operations like insert, update, delete and find. The

@TransactionConfiguration annotation with the property defaultRollback set to true indicates that the transaction will be rolled back at the end of the execution and no data will be persisted when the transaction ends.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ProductTest {
@Autowired
private SessionFactory sessionFactory;

@Test
public void testCRUD()

                                                {

Product p1 = new Product();
p1.setName(“A”);



Product p2 = new Product();
p2.setName(“B”);



sessionFactory.getCurrentSession().save(p1);
sessionFactory.getCurrentSession().save(p2);



p1.setName(“C”);
sessionFactory.getCurrentSession().merge(p1);
List<Product> list = sessionFactory.getCurrentSession().createQuery(“from
Product”).list(); Assert.assertEquals(2L, list.size());



sessionFactory.getCurrentSession().delete(p1);
```

```
List<Product> list2 = sessionFactory.getCurrentSession().createQuery("from Product").list(); Assert.assertEquals(1L, list2.size());

}

}
```

You will see lots of errors when writing the test class because the entity does not exist yet. The next step is to create the entity class. The entity is denoted using @Entity annotation and the table name if not provided is the same as the entity name. The identifier generation strategy is AUTO which picks up either identity column, sequence or table based on the underlying database. The @Column annotation can override the name of the field which is by default the same as the property name. The nullable property makes sure the field can be null or not and the length property denotes the size of the sphere. Once you write the entity, all the compilation errors in the test class should function and the tests should pass without any errors.

```
@Entity
@Table(name="PRODUCT")
public class Product {

private Integer id;
private String name;

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="ID")
public Integer getId() {
return id;

}

public void setId(Integer id) {
this.id = id;

}

@Column(name="NAME", nullable=false, length=100) public String
getName() {
return name;
```

}

public void setName(String name) {
this.name = name;

}

}

Table structure
DROP TABLE `product`;
CREATE TABLE `product` (
`ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT NULL,
PRIMARY KEY (`ID`)

);

Develop the data access tier

Next we will look at the data access tier. We will start with an uncomplicated structure of the data access test and write the test case for the find all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ProductDaoImplTest {
@Autowired
private ProductDao dao ;

@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size()); }

}
```

We will see how to write the corresponding data access operation of the above test. The @Repository annotation indicates the class is a data access operation object which is also eligible for component scanning. The class level

@Transactional annotation indicates all the methods of the class are transactional by nature. The Query instance is obtained by calling Session. createQuery () which also gives all the rows of the Product entity. The ProductDao is an interface containing the method signature of all these operations.

```
@Repository
@Transactional
public class ProductDaoImpl  implements ProductDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public List<Product> getAll() {
return sessionFactory.getCurrentSession().createQuery("select product from Product product order by product.id desc").list(); }

}
```

So far we have just seen the operation to get all the records. Let's look at the insert operation test. We insert a couple of rows into the Product table and check if the results match 2 records.

```
public class ProductDaoImplTest {
@Test
public void testInsert() {
Product sproduct = new Product();
sproduct.setName("Spring in Action Book");
dao.insert(sproduct);


Product hProduct = new Product();
hProduct.setName("Hibernate in Action Book"); dao.insert(hProduct);


Assert.assertEquals(2L, dao.getAll().size()); }

}
```

The corresponding insert operation in the data access class has to be implemented. Hibernate 4 Session. save () operation is implemented in the operation.

```
public class ProductDaoImpl  implements ProductDao {
public void insert(Product product) {
sessionFactory.getCurrentSession().save(product); }

}
```

We will jump to the next operation which is finding a row with an identifier. Please note all the code you write in a test case should be portable. In other words JUnit test cases should not have any order and can be executed in any sequence.

```
public class ProductDaoImplTest {
@Test
public void testGetById() {
Product sproduct = new Product();
sproduct.setName("Spring in Action Book");
dao.insert(sproduct);


List<Product> pList = dao.getAll();
Product product = pList.get(0);


Product product2 = dao.getById(product.getId()); Assert.assertEquals("Spring in Action Book", product2.getName()); }

}
```

The corresponding data access method is shown below. Hibernate 4 Session. get () operation is implemented in the operation.

```
public class ProductDaoImpl  implements ProductDao {
public Product getById(Integer id) {
return (Product) sessionFactory.getCurrentSession().get(Product.class, id); }

}
```

We will look at the delete operation writing the test first, followed by the data access operation. First, we insert 2 records followed by 1 delete and then check whether 1 record exists in the database.

```
public class ProductDaoImplTest {
```

```
@Test
public void testDelete() {
Product sproduct = new Product();
sproduct.setName(“Spring in Action Book”);
dao.insert(sproduct);


Product hProduct = new Product();
hProduct.setName(“Hibernate in Action Book”); dao.insert(hProduct);


Assert.assertEquals(2L, dao.getAll().size());
dao.delete(hProduct);


Assert.assertEquals(1L, dao.getAll().size()); }

                                                    }
```

The delete data access operation uses Hibernate 4 Session.delete () operation.

```
public class ProductDaoImpl implements ProductDao {
public void delete(Product product) {
sessionFactory.getCurrentSession().delete(product); }

                                                    }
```

Last, we will look at the update operation. The update test method inserts 1 row, updates the row, gets all the records, and checks that the name matches the update. Please note that here we could have retrieved the row using an identifier, but it would not have been portable. The code should not be hardcoded with an identifier because different systems will have a different sequence of execution resulting in different identifier values.

```
public class ProductDaoImplTest {
@Test
public void testUpdate() {
Product sproduct = new Product();
sproduct.setName(“Spring in Action Book”);
dao.insert(sproduct);
```

```
Assert.assertEquals(1L, dao.getAll().size());
List<Product> pList = dao.getAll();
Product product = pList.get(0);
product.setName(“Head First Design Patterns”);
dao.update(product);


List<Product> pList2 = dao.getAll();
Product product2 = pList2.get(0);
Assert.assertEquals(“Head First Design Patterns”, product2.getName()); }

                                                    }
```

The update data access operation is shown below. Hibernate 4 Session.merge () operation is used.

```
public class ProductDaoImpl  implements ProductDao {
public void update(Product product) {
sessionFactory.getCurrentSession().merge(product); }

                                                    }
```

Develop the business service tier

Now it is time to look at the business service tier. We will lead off with the test for fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ProductServiceImplTest {
@Autowired
private ProductService service;

@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size()); }

                                                    }
```

Let us look at how to code the corresponding method in the business service. The business service corresponding Spring annotation is @Service which is used along with the @Transactional annotation. The business service calls the data access tier, but remember that the data access tier uses entities and the business service tier uses data access objects. So there should exist a way to change the data access object to entities and vice versa. This is done via mappers which we will examine next.

```
@Service
@Transactional
public class ProductServiceImpl implements ProductService{
@Autowired
private ProductDao productDao;

@Autowired
private ProductMapper productMapper;

@Override
public List<ProductDto> findAll() {
List<Product> products = productDao.getAll(); List<ProductDto> productDtos
= new ArrayList<ProductDto>(); for(Product product : products){
productDtos.add(productMapper.mapEntityToDto(product)); }
return productDtos;

}

}
```

The mapper typically has two important operations which are used to convert data access objects to entities and vice versa. The @Component annotation helps in auto-wiring the component in the business service category as well as scanning the component using the auto-scan feature.

```
@Component
public class ProductMapper {

public Product mapDtoToEntity(ProductDto productDto){
Product product = new Product();
if(null!=productDto.getId()) product.setId(productDto.getId());
if(null!=productDto.getName()) product.setName(productDto.getName());
return product;
```

return product;

}

public ProductDto mapEntityToDto(Product product){
ProductDto productDto = new ProductDto();
if(null!=product.getId()) productDto.setId(product.getId());
if(null!=product.getName()) productDto.setName(product.getName()); return
productDto;

}

}

The create operation is next which is used to create a new row in the database. The test for the create operation is shown below which creates 2 instances of the entity and checks if the count is 2 with the fetch all records operation.
public class ProductServiceImplTest {
@Test
public void testCreate() {
ProductDto sproduct = new ProductDto();
sproduct.setName("Spring in Action Book");
service.create(sproduct);

ProductDto hProduct = new ProductDto();
hProduct.setName("Hibernate in Action Book"); service.create(hProduct);

Assert.assertEquals(2L, service.findAll().size()); }

}

The corresponding method in the business service tier accepts a data access object, converts to entity, and delegates to the data access tier to commit to the database.
public class ProductServiceImpl implements ProductService{
public void create(ProductDto productDto) {
productDao.insert(productMapper.mapDtoToEntity(productDto)); }

}

The next operation we will describe is finding a row with a given identifier. Always remember that while writing tests you cannot enforce an order and any test may be picked up for execution. So the test should work if executed in any random order.

```
public class ProductServiceImplTest {
@Test
public void testFindById() {
ProductDto sproduct = new ProductDto();
sproduct.setName(“Spring in Action Book”);
service.create(sproduct);


List<ProductDto> pList = service.findAll(); ProductDto product = pList.get(0);


ProductDto product2 = service.findById(product.getId());
Assert.assertEquals(“Spring in Action Book”, product2.getName()); }
```

}

The corresponding operation in business service to find a row with a given identifier will be described now. Again, the same sequence of contacting the data access tier and conversing with the mapper is followed.

```
public class ProductServiceImpl implements ProductService{
public ProductDto findById(Integer id) {
Product product = productDao.getById(id);
ProductDto productDto = null;
if(null !=product){
productDto = productMapper.mapEntityToDto(product); }
return productDto;
```

}

}

The remove operation is next in line which accepts a given identifier and removes the matching row from the database. The test method first creates two

cases, removes the first instance and checks whether one instance remains in the database.

```
public class ProductServiceImplTest {
@Test
public void testRemove() {
ProductDto sproduct = new ProductDto();
sproduct.setName(“Spring in Action Book”);
service.create(sproduct);


ProductDto hProduct = new ProductDto();
hProduct.setName(“Hibernate in Action Book”); service.create(hProduct);


Assert.assertEquals(2L, service.findAll().size());
List<ProductDto> pList = service.findAll(); ProductDto product = pList.get(0);
service.remove(product.getId());


Assert.assertEquals(1L, service.findAll().size()); }

                                                    }
```

The matching business service method is defined next which accepts an identifier and removes the matching instance from the database via the data access tier operation.

```
public class ProductServiceImpl implements ProductService{
public void remove(Integer productId) {
Product product = productDao.getById(productId); productDao.delete(product);

                                                    }

}
```

The last operation we will look at is the edit operation which takes a data access object and applies the data access method to merge the changes into the database. The test method creates an instance, edits the name and then checks if the edit was successful by comparing the name.

```
public class ProductServiceImplTest {
@Test
```

```
@Test
public void testEdit() {
ProductDto sproduct = new ProductDto();
sproduct.setName(“Spring in Action Book”);
service.create(sproduct);


Assert.assertEquals(1L, service.findAll().size());
List<ProductDto> pList = service.findAll(); ProductDto product = pList.get(0);
product.setName(“Head First Design Patterns”);
service.edit(product);


List<ProductDto> pList2 = service.findAll(); ProductDto product2 =
pList2.get(0);
Assert.assertEquals(“Head First Design Patterns”, product2.getName()); }

}
```

The equivalent operation of the business service is the edit operation which will be described below.

```
public class ProductServiceImpl implements ProductService{
public void edit(ProductDto productDto) {
productDao.update(productMapper.mapDtoToEntity( productDto )); }

}
```

Develop the presentation tier

The last tier in the stack is the REST based Spring Controller. First, we will look at how to write the test and then follow with the corresponding method in the controller. The @WebAppConfiguration is used to make sure the ApplicationContext is of the WebApplicationContext type. Gson is a Java library that can be used to convert Java Objects into their JSON representation. MockMvc instance is constructed from the WebApplicationContext. MockMvcRequestBuilders post and get methods are used to send the request.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
```

```
@TransactionConfiguration(defaultRollback = true) @Transactional
public class ProductControllerTest {

private Gson gson = new GsonBuilder().create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build(); }

@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                        }


public void testCreate() throws Exception {
ProductDto productDto = new ProductDto();
productDto.setName(“ABC”);

String json = gson.toJson(productDto);
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“standalonecreate”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}


public void testUpdate() throws Exception {
```

```
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("standalonefindAll"); MvcResult result =
this.mockMvc.perform(requestBuilder2).andReturn(); String response2 =
result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<ProductDto>>() {}.getType(); List<ProductDto>
productDtoList = gson.fromJson(response2, listType); ProductDto productDto2
= productDtoList.get(0); productDto2.setName("DEF");
String json2 = gson.toJson(productDto2);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("standaloneedit");
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

public void testDelete() throws Exception {


MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("standalonefindAll"); MvcResult result =
this.mockMvc.perform(requestBuilder2).andReturn(); String response2 =
result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<ProductDto>>() {}.getType(); List<ProductDto>
productDtoList = gson.fromJson(response2, listType); ProductDto productDto2
= productDtoList.get(0); MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("standaloneremove/"+productDto2.getId());
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

}
```

The REST Spring Controller will be contacting the business service tier to connect to the database. The @Controller annotation indicates the class serves the purpose of a controller with the @RequestMapping annotation used to map URLs as class-specific or method-specific. The @ResponseBody annotation indicates that the method return value is bound to the organic structure of the web response. Please note that the code is similar to the mock controller developed earlier, with a difference in the request mapping and the call going to

the actual database instead of the mock in-memory database.

```
@Controller
@RequestMapping(value=”/standalone”)
@Transactional
public class ProductController {

@Autowired
private ProductService service ;


@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<ProductDto> findAll(){
return service.findAll();

                                        }


@RequestMapping(value=”/findById/{productid}”,
method=RequestMethod.GET) public @ResponseBody ProductDto
findById(@PathVariable(“productid”) Integer productid){
return service.findById(productid);

                                        }


@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ProductDto product){
service.create(product);

                                        }


@RequestMapping(value=”/remove/{productid}”,
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT) public void remove(@PathVariable(“productid”)
Integer productid){
service.remove(productid);

                                        }
```

@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody ProductDto product){
service.edit(product);

}

}
We have come to the end of the chapter where we will discuss how to integrate the development work done by both the teams. The REST service will need to be integrated with the actual user interface. Please note that the request mapping changes from “*standalone*mock” to “/standalone”.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a standalone single table
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Part II. Managing One-to-One Relationships

Chapter 3. One-to-One Unidirectional Relationship

A one-to-one unidirectional relationship is a one-to-one relationship that does not refer back to the parent entity and hence is deemed to be unidirectional. In this chapter, we will follow the accustomed flow of starting with the user interface followed by service development, then join both modules at the culmination of the chapter. This approach will allow you to first master the user interface by taking an example of a related entity followed by the REST service development, so you will leave the chapter ready to fully execute a one-to-one unidirectional relationship.

Domain Model

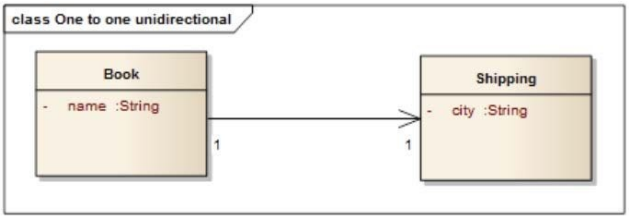

Figure 3-1: A one to one unidirectional relationship We will first take an example of a one-to-one unidirectional relationship and build all the tiers of the application on it. As seen in Figure 3-1, the Book entity has an identifier and a name field, which is a String field. Similarly, the Shipping entity has an identifier as well as a city field, which is also a String. In one instance of Book an instance of Shipping must be present. The Book is the owning side of the relationship from which Shipping can be traversed but not vice-versa. We will look at the entity design in the next sections of this chapter. As with the previous chapter, we will start by looking at the mock user interface before turning to the service development.

Develop a User Interface

We will divide the mock user interface development into three major tasks – the data transfer object, the mock service and the mock user interface. When the database service is ready, we will create the actual user interface and integrate it.

Develop the data transfer object

The data transfer object encapsulates the actual entity design and does not allow the user interface or the client to know the entity design underneath. Look at the data transfer object carefully and you will see the identifier of the Book object along with a name and city field. We are not exposing the Shipping entity or its identifier. We form a relationship between the Book and Shipping in the service, but to the user it is one object called Book. This is particularly useful for the user interface team as they do not have to manage the Shipping entity identifier and can just call the service as one object.

```
public class BookDto {
private Integer id;
private String name;
private String city;
```

```
// getters and setters

                                                    }
```

Develop the mock service

Now that the data transfer object is set up, let’s add some functionality. The in-memory database follows the Singleton pattern implemented with enum as a best practice. A list of BookDto is saved in memory where the add method inserts a new entry to the list. The identifier field starts with 1 and increments by 1 for every new entry.

```
public enum BookInMemoryDB {
INSTANCE;
private static List<BookDto> list = new ArrayList<BookDto>(); private static
Integer lastId = 0;
public Integer getId() {
return ++lastId;

                                                    }

public void add(BookDto bookDto){
bookDto.setId(getId());
list.add(bookDto);

                                                    }

                                                    }
```

The method create accepts an instance of BookDto passed from the user interface and adds an entry to the in memory list.

```
@Controller
@RequestMapping(value=”onetooneunidirectionalmock”) public class
BookMockController {
@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody BookDto bookDto){
BookInMemoryDB.INSTANCE.add(bookDto);
    }
```

}

Next, let’s look at the find all rows and find with identifier functionality.

```
public enum BookInMemoryDB {
public List<BookDto> findAll(){
return list;

}

public BookDto findById(Integer id) {
for(BookDto bookDto : list){
if(bookDto.getId() == id) return bookDto ; }
return null;

}

}
```

The find all rows returns a list of BookDto, and find with identifier returns an instance of BookDto.

```
@Controller
@RequestMapping(value=”onetooneunidirectionalmock”) public class
BookMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<BookDto> findAll(){
return BookInMemoryDB.INSTANCE.findAll();

}

@RequestMapping(value=”/findById/{bookid}”,
method=RequestMethod.GET) public @ResponseBody BookDto
findById(@PathVariable(“bookid”) Integer bookid){
return BookInMemoryDB.INSTANCE.findById(bookid); }

}
```

Subsequently, we will look at the remove operation which removes a specific row with an identifier.

```
public enum BookInMemoryDB {
public void remove(Integer id) {
BookDto dto = null;
```

```
BookDto dto = null;
for(BookDto bookDto : list){
if(bookDto.getId() == id){
dto = bookDto;
break;

                                                    }

                                                    }

if(null != dto){list.remove(dto);}

                                                    }

                                                    }
```

The remove operation deletes a specific row with the given identifier.

```
@Controller
@RequestMapping(value="/onetooneunidirectionalmock") public class
BookMockController {
@RequestMapping(value="/remove/{bookid}", method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT) public void
remove(@PathVariable("bookid") Integer bookid){
BookInMemoryDB.INSTANCE.remove(bookid);

                                                    }

                                                    }
```

Last we look at the edit method which updates a row.

```
public enum BookInMemoryDB {
public void edit(BookDto bookDto){
for(BookDto dto : list){
if(dto.getId() == bookDto.getId()){
dto.setName(bookDto.getName());
dto.setCity(bookDto.getCity());

                                                    }

                                                    }
```

}

}

The controller edit operation updates a row with details. The BookDto is passed as a parameter from the user interface.

```
@Controller
@RequestMapping(value="onetooneunidirectionalmock") public class
BookMockController {
@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody BookDto bookDto){
BookInMemoryDB.INSTANCE.edit(bookDto);
```

}

}

Develop the mock user interface

In the previous section, we saw how to develop the mock service with an in-memory database. Now, we will look at the user interface which will consume this mock service as shown in Figure 3-2. We will be using JSP for the user interface and JQuery with AJAX to connect to the mock service.

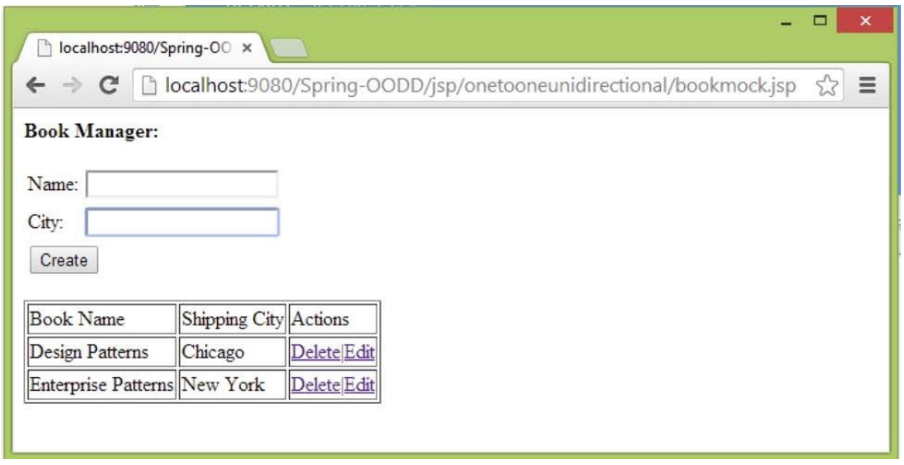

Figure 3-2: Book and Shipping User Interface Let's start by looking at the body of the JSP page, which mentions the Javascript methods called. We will describe the flow with the Javascript calls.

```
<body onload=”loadObjects()”>
<div id=”container”>
<div>
<p>
<b>Book Manager:</b>
</p>
</div>
<form method=”post” id=”bookForm”>
<table>
<tbody>
<tr>
<td>Name:</td>
<td><input name=”bookId” id=”bookId” type=”hidden”> <input
name=”bookname” id=”bookname” type=”text”></td> </tr>
<tr>
<td class=”tdLabel”><label for=”ad002_dob” class=”label”>City:</label></td>
<td><input type=”text” name=”city” value=”” id=”city” ><td> </tr>
<tr>
<td colspan=”2”><input type=submit value=”Create”
id=”subButton” onclick=”return methodCall()”></td> </tr>
</tbody>
</table>
</form>
<div id=”bookFormResponse”></div> </div>
</body>
```

First, note the JavaScript loadObjects method that is called when the page is loaded. The URL points to the REST mock service we developed earlier, which loads all the rows. When the function is successful, another JavaScript method processResponseData is called.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDonetooneunidirectional/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR) {
processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“bookname”).value=””; alert(“Error Status Load
```

```
Objects:”+textStatus); }

});

return false;

}
```

The processResponseData JavaScript method called when the above service call is successful is shown below. The table is populated with the contents of the in-memory database and the links to the edit and delete the row entries are specified.

```
function processResponseData(responsedata){
var dyanamicTableRow=”<table border=1>”+
“<tr>” +
“<td>BookName</td>”+”<td>Shipping City</td>”+
“<td>Actions</td>”+
“</tr>”;
var dataRow=””;
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue); });
dyanamicTableRow=dyanamicTableRow+dataRow+”</table>”;
document.getElementById(“bookFormResponse”).innerHTML=dyanamicTableR
}
function generateTableData(itemvalue){
var dataRow=”<tr>” +
“<td>” +itemvalue.name+”</td>”+
“<td>”+itemvalue.city+”</td>”+
“<td>” +
“<a href=# onclick=deleteObject(“+itemvalue.id+”)>Delete</a>”+
“|<a href=# onclick=editObject(“+itemvalue.id+”)>Edit</a>”+
“</td>”+
“</tr>”;
return dataRow;

}
```

Now that we have observed how the page loading works, let’s look at the create functionality. If you look at the JSP fragment you will see a Javascript method methodCall is being called. This method works for both create and update

functionality.

```
function methodCall(){
var buttonValue = document.getElementById(“subButton”).value;
if(buttonValue==”Create”){
create();
}else if(buttonValue==”Update”){
update();

                                                    }

return false;

                                                    }
```

First, look at create where the name of the book and shipping city is passed as an argument embedded in the BookDto. The JSON object is formed by taking the values from the user interface and passed to the mock service as a POST parameter. If successful, the values are reset in the user interface and if there are any errors, a notification is shown to the user.

```
function create(){
var name = $(“#bookname”).val();
var city = $(“#city”).val();
var formData={“name”:name,”city”:city};
$.ajax({
url : “Spring-OODDonetooneunidirectional/mock/create”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR) {
document.getElementById(“bookname”).value=””;
document.getElementById(“city”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“bookname”).value=””; alert(“Error Status
Create:”+textStatus); }

                                                    });
```

return false;

}

Next we will look at edit functionality where the book name and identifier as well as the shipping city are being passed as an argument. The update function is similar to the create function with the difference that the id is part of the JSON object formed and hits the edit mock service.

```
function update(){
var name = $("#bookname").val();
var id = +$("#bookId").val();
var city = $("#city").val();
var formData={"id":id,"name":name,"city":city}; $.ajax({
url : "Spring-OODDonetooneunidirectional/mock/edit", type: "POST",
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR) {
document.getElementById("bookname").value="";
document.getElementById("city").value="";
document.getElementById("subButton").value="Create"; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("bookname").value=""; alert("Error Status
Update:"+textStatus); }
```

});

return false;

}

The Javascript methods editObject and viewObject allow the selection of a row for the edit operation open to the user.

```
function editObject(bookId){
editurl="Spring-OODDonetooneunidirectional/mock/findById/"+bookId; var
bookForm={id:bookId};
$.ajax({
url : editurl,
```

```
type: “GET”,
data : bookForm,
dataType: “json”,
success: function(data, textStatus, jqXHR) {
viewObject(data);
document.getElementById(“subButton”).value=”Update”;
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Find Object:”+textStatus); }

                                                    });

                                                     }

function viewObject(data){
document.getElementById(“bookname”).value=data.name;
document.getElementById(“city”).value=data.city;
document.getElementById(“bookId”).value=data.id; }
```
Finally, look at the delete operation where the book identifier is passed as an argument. Once a delete is successful the table containing the data is refreshed using AJAX.
```
function deleteObject(bookId){
var bookForm ={id:bookId};
delurl=”Spring-OODDonetooneunidirectional/mock/remove/”+bookId; $.ajax({
url : delurl,
type: “POST”,
data : bookForm,
dataType: “json”,
success: function(data, textStatus, jqXHR) {
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus); }

                                                    });

                                                     }
```
Develop the Service

There are four major steps in service development– creating the entity, creating the data access tier, creating the business service tier and creating the JSON-based REST service, which will be integrated with the user interface. We will follow a bottom-up approach here, starting with the entity and going up to the REST service.

Develop the resource (entity) tier Following a test-driven development approach, we will write the test for each method followed by the definition of the actual method. In the entity test method, first create two instances of book entity followed by an update and a delete, and then finally check whether one instance of the book remains. After the test executes, the operation is rolled back.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class BookTest {
@Autowired
private SessionFactory sessionFactory;
@Test
public void testCRUD()

                                                {

Book book1 = new Book();
book1.setName(“Java SE”);
Shipping shipping1 = new Shipping();
shipping1.setCity(“US”);
book1.setShipping(shipping1);
sessionFactory.getCurrentSession().save(book1);
Book book2 = new Book();
book2.setName(“EJB 3.0”);
Shipping shipping2 = new Shipping();
shipping2.setCity(“CAN”);
```

```
shipping2.setCity("CAN");
book2.setShipping(shipping2);
sessionFactory.getCurrentSession().save(book2);
book1.setName("JEE");
shipping1 = book1.getShipping();
shipping1.setCity("UK");
sessionFactory.getCurrentSession().merge(book1);
List<Book> books = sessionFactory.getCurrentSession().createQuery("select
book from Book book").list(); Assert.assertEquals(2L, books.size());


sessionFactory.getCurrentSession().delete(book2);
   books = sessionFactory.getCurrentSession().createQuery("select book from
Book book").list(); Assert.assertEquals(1L, books.size());

                                                    }

                                                    }
```

We are developing a one-to-one unidirectional relationship where Book is the owner and Shipping is the counterpart, which can be accessed from the Book. First, look at the Shipping entity, which does not have any dependents.

```
@Entity
@Table(name="SHIPPING")
public class Shipping {
private Integer id;
private String city;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="id",length=20)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }
```

```
@Column(name="city",length=50)
public String getCity() {
return city;

                                                    }

public void setCity(String city) {
this.city = city;

                                                    }

                                                    }
```

The next entity is the Book, which contains a @OneToOne reference to Shipping. Please note that in an instance of Book a corresponding instance of Shipping exists. The Shipping instance is created first and then stored as a reference in the Book entity. When a target entity in a one-to-one or one-to-many relationship is removed from the relationship, it is often desirable to cascade the remove operation to the target entity. Such target entities are considered "orphans," and the orphanRemoval attribute can be used to specify that orphaned entities should be removed. For example, if the shipping entity is changed, then the previous entity attached to the Book is considered an orphan which will be removed.

```
@Entity
@Table(name="BOOK")
public class Book {
private Integer id;
private String name;
private Shipping shipping;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="id",length=20)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }
```

```
}

@Column(name="name",length=50)
public String getName() {
return name;

}

public void setName(String name) {
this.name = name;

}

@OneToOne(cascade=CascadeType.ALL, orphanRemoval=true)
@JoinColumn(name="shipping_id")
public Shipping getShipping() {
return shipping;

}

public void setShipping(Shipping shipping) {
this.shipping = shipping;

}

}
```

Table structure

```
DROP TABLE `book`;
DROP TABLE `shipping`;
CREATE TABLE `shipping` (
  `id` int(11) NOT NULL AUTO_INCREMENT, `city` varchar(50) DEFAULT
NULL, PRIMARY KEY (`id`)

);

CREATE TABLE `book` (
  `id` int(11) NOT NULL AUTO_INCREMENT, `name` varchar(50)
DEFAULT NULL, `shipping_id` int(11) DEFAULT NULL, PRIMARY KEY
```

```
(`id`),
  KEY (`shipping_id`),
  CONSTRAINT FOREIGN KEY (`shipping_id`) REFERENCES `shipping`
(`id`) );
```

Develop the data access tier

Next we will look at the data access tier. We will start with the test case for the find all records operation. Since this is the first function which is being developed as part of test-driven development, we let it return zero size for the find all records operation. In the functions that will follow we will build the CRUD operations in sequence.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class BookDaoImplTest {

@Autowired
private BookDao dao;

@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size()); }

                                                    }
```

We will see how to write the conforming data access operation for the test above. The BookDao is an interface containing the method signature of all these operations. The Hibernate Session.createQuery operation is used with a SQL to retrieve all the records from the Book entity.

```
@Repository
@Transactional
public class BookDaoImpl implements BookDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public List<Book> getAll() {
return (List<Book>) sessionFactory.getCurrentSession() .createQuery(“select
```

book from Book book order by book.id desc").list(); }

}

Let's look at the insert operation test. Insert a few rows into the Book table and check if the results match two records. Note that the corresponding Shipping entity also gets created during the insert operation of the Book entity.

```
public class BookDaoImplTest {

@Autowired
private BookDao dao;

@Test
public void testInsert(){
Book book1 = new Book();
book1.setName("Java Book");
Shipping shipping1 = new Shipping();
shipping1.setCity("UK");
book1.setShipping(shipping1);
dao.insert(book1);

Book book2 = new Book();
book2.setName("Java SE Book");
Shipping shipping2 = new Shipping();
shipping2.setCity("IN");
book2.setShipping(shipping2);
dao.insert(book2);

Assert.assertEquals(2L, dao.getAll().size()); }

}
```

The corresponding insert operation just calls the Hibernate Session.save method. The Shipping entity identifier gets created automatically along with the identifier of the Book entity.

```
public class BookDaoImpl implements BookDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public void insert(Book book) {
```

```
public void insert(Book book) {
sessionFactory.getCurrentSession().save(book); }
}
```

The next operation we will look at is finding a row with an identifier. When writing test cases related to identifier, one must be careful not to hard code any identifier, but to make the code portable by retrieving identifier using find all records operation.

```
public class BookDaoImplTest {

@Autowired
private BookDao dao;

@Test
public void testGetById() {
Book book1 = new Book();
book1.setName("Java Book");
Shipping shipping1 = new Shipping();
shipping1.setCity("UK");
book1.setShipping(shipping1);
dao.insert(book1);



List<Book> books = dao.getAll();
Book book = books.get(0);



Book tempbook = dao.getById(book.getId());
Assert.assertEquals(tempbook.getName(),book.getName());

                                                    }

                                                    }
```

The corresponding data access method which uses Hibernate 4 Session.get () operation is shown.

```
public class BookDaoImpl implements BookDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public Book getById(int id) {
```

```
public Book getById(int id) {
return (Book) sessionFactory.getCurrentSession().get(Book.class,id); }

}
```

Now that the test has been written, let's look at the delete operation writing the data access operation. First, insert two records followed by a delete and then test whether a single record exists in the database.

```
public class BookDaoImplTest {

@Autowired
private BookDao dao;

@Test
public void testDelete() {
Book book1 = new Book();
book1.setName("Java Book");
Shipping shipping1 = new Shipping();
shipping1.setCity("UK");
book1.setShipping(shipping1);
dao.insert(book1);

Book book2 = new Book();
book2.setName("Java SE Book");
Shipping shipping2 = new Shipping();
shipping2.setCity("IN");
book2.setShipping(shipping2);
dao.insert(book2);

Assert.assertEquals(2L, dao.getAll().size());
dao.delete(book2);
Assert.assertEquals(1L, dao.getAll().size());

}

}
```

The delete operation uses the Hibernate 4 Session.delete () method shown below.

```
public class BookDaoImpl implements BookDao {
```

```
@Autowired
private SessionFactory sessionFactory ;
@Override
public void delete(Book book) {
  sessionFactory.getCurrentSession().delete(book); }

                                                    }
```

Last, look at the update test operation which inserts a row, updates the row, gets all the records, and tests whether the name matches. The code should not be hard coded with an identifier because different systems will have a different sequence of execution resulting in different identifier values.

```
public class BookDaoImplTest {

@Autowired
private BookDao dao;
@Test
public void testUpdate() {
Book book1 = new Book();
book1.setName(“Java Book”);
Shipping shipping1 = new Shipping();
shipping1.setCity(“UK”);
book1.setShipping(shipping1);
dao.insert(book1);


Assert.assertEquals(1L, dao.getAll().size());
List<Book> books = dao.getAll();
Book book2 = books.get(0);


Shipping shipping = book2.getShipping();
shipping.setCity(“IND”);
dao.update(book2);


List<Book> books1 = dao.getAll();
Book book3 = books1.get(0);
Assert.assertEquals(“IND”, book3.getShipping().getCity()); }
```

}

The update data access operation is shown below. It uses Hibernate 4 Session.merge () operation.

```
public class BookDaoImpl implements BookDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public void update(Book book) {
sessionFactory.getCurrentSession().merge(book); }
```

}

Develop the business service tier

We will begin with the business service tier as the data access tier has been developed and is ready for integration with the business service tier. First, let's look at the test for fetching all records operation. Since this is the first function being tested, the other operations are still not developed and only the find all records operation can be tested with zero records returned when the table is empty.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class BookServiceImplTest {
@Autowired
private BookService service ;
@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size()); }
```

}

Look at how to code the matching method in the business service. The data access tier uses entities and the business service tier uses data access objects. So there exists a mapper which transforms a data transfer object to an entity and vice-versa.

```
@Service
```

```
@Transactional
public class BookServiceImpl implements BookService {
@Autowired
private BookDao dao;
@Autowired
private BookMapper mapper;
@Override
public List<BookDto> findAll() {
List<Book> books = dao.getAll();
List<BookDto> bookDtos = new ArrayList<BookDto>(); for(Book book :
books){
bookDtos.add(mapper.mapEntityToDto(book)); }
return bookDtos;

                                                    }

                                                    }
```

The mapper typically has two significant operations, which transform a data access object to its equivalent entity and vice-versa. One thing to observe here is that the Shipping identifier is created every time the Book is updated or added. As we have mentioned before, the management of the Shipping entity identifier will be done internally by the service. The user interface team will only pass the data values in the Shipping entity, like city field, and is not bothered about the identifier of the Shipping entity.

```
@Component
public class BookMapper {
public Book mapDtoToEntity(BookDto bookDto){
Book book = new Book();
Shipping shipping = new Shipping();
if(null!=bookDto.getId() book.setId(bookDto.getId());
if(null!=bookDto.getName()) book.setName(bookDto.getName());
if(null!=bookDto.getCity()){ shipping.setCity(bookDto.getCity());
book.setShipping(shipping);}
return book;

                                                    }

public BookDto mapEntityToDto(Book book){
BookDto bookDto = new BookDto();
```

```
if(null!=book.getId()) bookDto.setId(book.getId()); if(null!=book.getName())
bookDto.setName(book.getName()); if(null!=book.getShipping()){
if(null !=book.getShipping().getCity()){
bookDto.setCity(book.getShipping().getCity()); }

                                                    }

return bookDto;

                                                    }

                                                    }
```

The create test operation follows which creates an instance of the entity and checks if the count is one with the fetch all records operation.

```
public class BookServiceImplTest {
@Autowired
private BookService service ;
@Test
public void testCreate() {
BookDto bookDto = new BookDto();
bookDto.setName(“Java SE”);
bookDto.setCity(“IND”);
service.create(bookDto);


Assert.assertEquals(1L, service.findAll().size()); }

                                                    }
```

The agreeing method in the business service tier accepts a data access object, converts to the equivalent entity and delegates to the data access tier for saving to the database.

```
public class BookServiceImpl implements BookService {
@Autowired
private BookDao dao;
@Autowired
private BookMapper mapper;
@Override
public void create(BookDto bookDto) {
```

```
public void create(BookDto bookDto) {
dao.insert(mapper.mapDtoToEntity(bookDto)); }

}
```

The following test operation we will define as finding a row with a given identifier.

```
public class BookServiceImplTest {
@Autowired
private BookService service ;
@Test
public void testFindById() {
BookDto bookDto = new BookDto();
bookDto.setName(“Java SE”);
bookDto.setCity(“IND”);
service.create(bookDto);


List<BookDto> bookDtos = service.findAll(); BookDto bDto = bookDtos.get(0);


BookDto bDto2 = service.findById(bDto.getId()); Assert.assertEquals(“Java SE”, bDto2.getName()); Assert.assertEquals(“IND”, bDto2.getCity()); }

}
```

The analogous operation of business service to find a row with a given identifier will now be described. Yet again the same order is followed: contacting the data access tier and conversing with the mapper.

```
public class BookServiceImpl implements BookService {
@Autowired
private BookDao dao;
@Autowired
private BookMapper mapper;
@Override
public BookDto findById(int id) {
Book book = dao.getById(id);
if(null !=book){
return mapper.mapEntityToDto(book);
```

}

return null;

}

}

The remove test operation is then in line which accepts a given identifier, and removes the equivalent row from the database. The test method first creates one instance, removes the same instance and checks whether there are zero results in the database.

```
public class BookServiceImplTest {
@Autowired
private BookService service ;
@Test
public void testRemove() {
BookDto bookDto = new BookDto();
bookDto.setName(“Java SE”);
bookDto.setCity(“IND”);
service.create(bookDto);


Assert.assertEquals(1L, service.findAll().size()); List<BookDto> bookDtos =
service.findAll(); BookDto bDto = bookDtos.get(0);


service.remove(bDto.getId());
Assert.assertEquals(0L, service.findAll().size()); }
```

}

The equivalent business service method is defined afterward, which accepts an identifier and removes the matching instance from the database through the data access tier operation.

```
public class BookServiceImpl implements BookService {
@Autowired
private BookDao dao;
@Autowired
```

```
private BookMapper mapper;
@Override
public void remove(int id) {
Book book = dao.getById(id);
dao.delete(book);

                                                    }

}
```

The last test operation is the edit which creates an instance, edits the city and then checks if the edit was effective.

```
public class BookServiceImplTest {
@Autowired
private BookService service ;
@Test
public void testEdit() {
BookDto bookDto = new BookDto();
bookDto.setName(“Java SE”);
bookDto.setCity(“IND”);
service.create(bookDto);


Assert.assertEquals(1L, service.findAll().size());
List<BookDto> bookDtos2 = service.findAll(); BookDto bDto2 =
bookDtos2.get(0);
bDto2.setCity(“CAN”);
service.edit(bDto2);


List<BookDto> bookDtos = service.findAll(); BookDto bDto = bookDtos.get(0);


Assert.assertEquals(“CAN”, bDto.getCity()); }

                                                    }
```

The equivalent operation of the business service is the edit operation which deputies the call to the data access update method after conversion with the mapper method. Please note that the edit operation does not have the identifier of

the Shipping entity and hence generates a new instance.

```
public class BookServiceImpl implements BookService {
@Autowired
private BookDao dao;
@Autowired
private BookMapper mapper;
@Override
public void edit(BookDto bookDto) {
dao.update(mapper.mapDtoToEntity(bookDto)); }

}
```

Develop the presentation tier

The final tier in the stack is the REST-based Spring Controller for which we will first write the test. The test first creates an instance of BookDto. Next it retrieves that instance using find all records operation and edits the values of the instance. Lastly the instance is deleted via the REST service.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class BookControllerTest {
private Gson gson = new GsonBuilder().create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;
@Before
public void setUp() {
mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build(); }
@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

}
```

```
public void testCreate() throws Exception {
BookDto bookDto = new BookDto();
bookDto.setName(“Java SE”);
bookDto.setCity(“IND”);
String json = gson.toJson(bookDto);
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“onetooneunidirectionalcreate”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}
public void testUpdate() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get(“onetooneunidirectionalfindAll”); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<BookDto>>() {}.getType(); List<BookDto> bookDtos =
gson.fromJson(response2, listType); BookDto bookDto2 = bookDtos.get(0);
bookDto2.setName(“JEFF”);
String json2 = gson.toJson(bookDto2);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post(“onetooneunidirectionaledit”);
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}
public void testDelete() throws Exception {
MockHttpServletRequestBuilder requestBuilder =
MockMvcRequestBuilders.get(“onetooneunidirectionalfindAll”); MvcResult
result = this.mockMvc.perform(requestBuilder).andReturn(); String response2 =
result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<BookDto>>() {}.getType(); List<BookDto> bookDtos =
gson.fromJson(response2, listType); BookDto bookDto2 = bookDtos.get(0);
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.post(“onetooneunidirectionalremove/”+bookDto2.get
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
}
```

}

The approving REST Spring Controller will be contacting the business service tier which is finally through the data access tier to connect to the database. Here the code is similar to the mock service, with the difference that the code is hitting the actual database and not the in memory one.

```
@Controller
@RequestMapping(value=”/onetooneunidirectional”) @Transactional
public class BookController {
@Autowired
private BookService service ;


@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<BookDto> findAll(){
return service.findAll();

}

@RequestMapping(value=”/findById/{bookId}”,
method=RequestMethod.GET) public @ResponseBody BookDto
findById(@PathVariable(“bookId”) Integer bookId){
return service.findById(bookId);

}

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody BookDto book){
service.create(book);

}

@RequestMapping(value=”/remove/{bookId}”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT) public void
remove(@PathVariable(“bookId”) Integer bookId){
service.remove(bookId);

}
```

```
@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody BookDto book){
service.edit(book);

}

}
```

We will now discuss how to integrate the REST service with the actual user interface. Please note that the request mapping changes from "*onetooneunidirectional*mock" to "/onetooneunidirectional." The rest of the work involves creating the actual user interface from the mock user interface.

Summary

In this chapter, we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a one-to-one unidirectional association
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 4. One-to-One Bidirectional Relationship

Relationship A one-to-one bidirectional relationship is a one-to-one relationship which does refer back both ways from parent to child and vice versa. If you find a real life situation where the relationship is a one to one and you need to traverse in both directions, use a one-to-one bidirectional relationship. We will monitor the habituated flow of starting with the user interface tracked by the service development and seam in both the modules at the conclusion of the chapter.

Domain Model

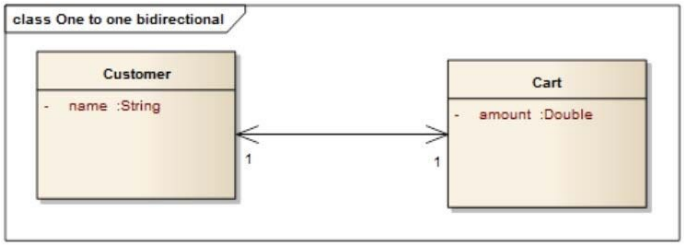

Figure 4-1: A one-to-one bidirectional relationship

As shown in the one-to-one bidirectional association in Figure 4-1, the Customer

entity has an identifier and a String name field. The Cart entity also has an identifier along with a Double amount field. Because of the bidirectional nature of the relationship, in an instance of Customer an instance of Cart must be present and vice-versa. We will first write the mock user interface, then turn to the service. To develop the mock UI, let's look at the data transfer object, the mock service and the mock user interface.

Develop a User Interface

There are usually three major tasks we look at during the development of the mock user interface. They are the data transfer object, the mock service and the mock user interface.

Develop the data transfer object

Here the data transfer object does not have the Cart entity. Since the Customer and Cart share a one-to-one relationship, in an instance of Customer there can be only one instance of the Cart. We are hiding from the user that we are saving the relationship as a one-to-one bidirectional relationship.

```
public class CustomerDto {
private Integer id;
private String name;
private double amount;

    // getters and setters
}
```

Develop the mock service

Let's start with the add feature. We have a list of CustomerDto and with every new addition, one instance of CustomerDto is added to the list. Singleton pattern using enum is followed for the in-memory database implementation. The identifier field starts with 1 and increments by 1 for every new addition.

```
public enum CustomerInMemoryDB {
INSTANCE;
private static Integer lastId = 0;
private static List<CustomerDto> list = new ArrayList<CustomerDto>(); public
```

```
Integer getId() {
return ++lastId;

                                                    }

public void add(CustomerDto customerDto){
customerDto.setId(getId());
list.add(customerDto);

                                                    }

                                                    }
```

Now let’s look at the matching mock service implementation. The create method receives an instance of CustomerDto from the user interface and adds to the list via the mock database.

```
@Controller
@RequestMapping(value=”onetoonebidirectionalmock”) public class
CustomerMockController {
@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody CustomerDto customerDto){
CustomerInMemoryDB.INSTANCE.add(customerDto);

                                                    }

                                                    }
```

Next we observe the find all rows and find with identifier feature.

```
public enum CustomerInMemoryDB {
public List<CustomerDto> findAll(){
return list;

                                                    }

public CustomerDto findById(Integer id) {
for(CustomerDto customerDto : list){
if(customerDto.getId() == id) return customerDto ; }
return null;

                                                    }
```

}

The find all rows yields a list of CustomerDto and find with identifier gets an instance of CustomerDto.

```
@Controller
@RequestMapping(value="onetoonebidirectionalmock") public class
CustomerMockController {
@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody List<CustomerDto> findAll(){
return CustomerInMemoryDB.INSTANCE.findAll();

}

@RequestMapping(value="/findById/{customerid}",
method=RequestMethod.GET) public @ResponseBody CustomerDto
findById(@PathVariable("customerid") Integer customerid){
return CustomerInMemoryDB.INSTANCE.findById(customerid); }

}
```

Subsequently, we will implement the remove operation which removes a specific row with a given identifier.

```
public enum CustomerInMemoryDB {
public void remove(Integer id) {
CustomerDto dto = null;
for(CustomerDto customerDto : list){
if(customerDto.getId() == id){dto = customerDto;break;}

}

if(null !=dto){list.remove(dto);}

}

}
```

In the matching mock service, the remove operation deletes a specific row with the given identifier passed as an input to the URL.

```
@Controller
```

@RequestMapping(value="*onetoonebidirectional*mock") public class CustomerMockController {
@RequestMapping(value="/remove/{customerid}", method=RequestMethod.POST) @ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@PathVariable("customerid") Integer customerid){
CustomerInMemoryDB.INSTANCE.remove(customerid); }

}

Last we take care of the edit operation responsible for updating a row.
public enum CustomerInMemoryDB {
public void edit(CustomerDto customerDto){
for(CustomerDto dto : list){
if(dto.getId() == customerDto.getId()){
dto.setName(customerDto.getName());
dto.setAmount(customerDto.getAmount());

}

}

}

}

The controller edit operation accepts an edited copy of the CustomerDto and updates the specific row with the changes.
@Controller
@RequestMapping(value="*onetoonebidirectional*mock") public class CustomerMockController {
@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody CustomerDto customerDto){
CustomerInMemoryDB.INSTANCE.edit(customerDto); }

}

Develop the mock user interface

Let’s turn from developing the mock service with an in-memory database to the user interface, which will ingest this mock service as shown in Figure 4-2. The user interface will have ways to add, edit, delete and show all records related to the Customer including Cart amount field.

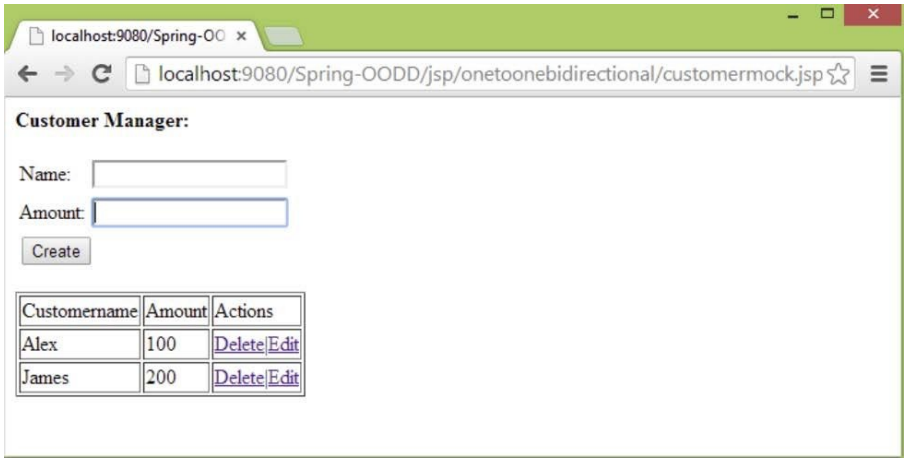

Figure 4-2: A screenshot of the Customer and Cart user interface The JSP body is the starting page which gets loaded and subsequently the JavaScript methods get called. We will define the flow with the JavaScript calls.

```
<body onload=”loadObjects()”>
<div id=”container”>
<div>
<p>
<b>Customer Manager:</b>
</p>
</div>
<form method=”post” id=”customerForm”>
<table>
<tbody>
<tr>
<td>Name:</td>
<td><input name=”customerid” id=”customerid” type=”hidden”> <input
name=”customername” id=”customername” type=”text”></td> </tr>
<tr>
<td class=”tdLabel”><label>Amount:</label></td> <td><input type=”text”
name=”amount” value=”” id=”amount” ><td> </tr>
<tr>
<td colspan=”2”><input type=submit value=”Create”
```

id=”subButton” onclick=”return methodCall()”></td> </tr>
</tbody>
</table>
</form>

</body>
First let’s look at the JavaScript loadObjects method, which is called when the page is loaded with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDonetoonebidirectional/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“customername”).value=””; alert(“Error Status Load Objects:”+textStatus); }

});

return false;

}
```

The processResponseData JavaScript method called when the above service call is successful is revealed in the following code. The table is occupied with the contents of the in-memory database, and the generateTableData JavaScript method is shown, which generates the table structure.

```
function processResponseData(responsedata){
var dyanamicTableRow=”<table border=1>”+
“<tr>” +
“<td>Customername</td>”+”<td>Amount</td>”+
“<td>Actions</td>”+
“</tr>”;
var dataRow=””;
```

```
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue);

                                                    });

dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("customerFormResponse").innerHTML=dyanamicTa
}
function generateTableData(itemvalue){
var dataRow="<tr>" +
"<td>" +itemvalue.name+"</td>"+
"<td>"+itemvalue.amount+"</td>"+
"<td>" +
"<a href=# onclick=deleteObject("+itemvalue.id+")>Delete</a>"+
"|<a href=# onclick=editObject("+itemvalue.id+")>Edit</a>"+
"</td>"+
"</tr>";
return dataRow;

                                                    }
```

We have seen the page loading mechanism. If you see the JSP fragment you will realize JavaScript method methodCall is being called which works for both create and update functionality.

```
function methodCall(){
var buttonValue = document.getElementById("subButton").value;
if(buttonValue=="Create"){
create();
}else if(buttonValue=="Update"){
update();

                                                    }

return false;

                                                    }
```

First, we will see the create JavaScript method, which takes the input from the text boxes name and amount and passes the input as a CustomerDto without an identifier.

```
function create(){
```

function create(){
var name = $(“#customername”).val();
var amount=$(“#amount”).val();
var formData={“name”:name,”amount”:amount};
$.ajax({
url : “*Spring-OODD*onetoonebidirectional/mock/create”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

document.getElementById(“customername”).value=””;
document.getElementById(“amount”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“customername”).value=””; alert(“Error Status Create:”+textStatus);

}

});

return false;

}

Next, we will look at the update JavaScript method, which updates a CustomerDto including an identifier, name and amount.
function update(){
var name = $(“#customername”).val();
var id = +$(“#customerid”).val();
var amount = +$(“#amount”).val();
var formData={“id”:id,”name”:name,”amount”:amount}; $.ajax({
url : “*Spring-OODD*onetoonebidirectional/mock/edit”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);

xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)

{

document.getElementById("customername").value="";
document.getElementById("amount").value="";
document.getElementById("subButton").value="Create"; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("customername").value="";
document.getElementById("amount").value="";
alert("Error Status Update:"+textStatus);

}

});

return false;

}

We will also look at the Javascript methods editObject and viewObject which show a record to be edited.
function editObject(customerid){
editurl="*Spring-OODD*onetoonebidirectional/mock/findById/"+customerid; var customerForm={id:customerid};
$.ajax({
url : editurl,
type: "GET",
data : customerForm,
dataType: "json",
success: function(data, textStatus, jqXHR)

{

viewObject(data);
document.getElementById("subButton").value="Update";
},
error: function (jqXHR, textStatus, errorThrown) {

error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Find Object:”+textStatus); }

});

}

function viewObject(data){
document.getElementById(“customername”).value=data.name;
document.getElementById(“amount”).value=data.amount;
document.getElementById(“customerid”).value=data.id; }
Lastly, we will look at the delete operation where the customer identifier is passed as an input and the related Customer information is removed and refreshed.
function deleteObject(customerid){
var customerForm={id:customerid};
delurl=”*Spring-OODD*onetoonebidirectional/mock/remove/”+customerid;
$.ajax({
url : delurl,
type: “POST”,
data : customerForm,
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus);

}

});

}

Develop the Service

Let’s now turn to the service development, which consists of four major steps –

creating the entity, creating the data access tier, creating the business service tier and creating the JSON-based REST. We will start with the entity and go up to the REST service.

Develop the resource (entity) tier Following a test-driven development approach, we will write the test for the entity followed by the entity implementation. First, create two instances of Customer with Cart entity followed by an update and a delete, and finally test whether one instance of Customer remains. Note that the essential difference is to populate the Cart for a Customer and also the Customer for a Cart both ways in a bidirectional relationship.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class CustomerTest {
@Autowired
private SessionFactory sessionFactory;
@Test
public void testCRUD()

                                                {

Customer customer1 = new Customer();
customer1.setName(“Alex”);
Cart cart1 = new Cart();
cart1.setAmount(500.0);
customer1.setCart(cart1);
sessionFactory.getCurrentSession().save(cart1);
Customer customer2 = new Customer();
customer2.setName(“Fred”);
Cart cart2 = new Cart();
cart2.setAmount(700.0);
```

```
customer2.setCart(cart2);
sessionFactory.getCurrentSession().save(customer2);



customer1.setName(“Alex”);
cart1 = customer1.getCart();
cart1.setAmount(500.0);
customer1.setCart(cart1);
sessionFactory.getCurrentSession().merge(customer1);
List<Customer> customers = sessionFactory.getCurrentSession()
.createQuery(“select customer from Customer customer order by customer.id
desc”).list(); Assert.assertEquals(2L, customers.size());



sessionFactory.getCurrentSession().delete(customer2);
customers = sessionFactory.getCurrentSession() .createQuery(“select customer
from Customer customer order by customer.id desc”).list();
Assert.assertEquals(1L, customers.size());

                                                    }

                                                    }
```

We are developing a one-to-one bidirectional relationship where Customer is the owner and Cart is the counterpart, so the relationship can be traversed both ways, meaning Cart from Customer and Customer from Cart. First, look at the owner (Customer containing a @OneToOne instance of Cart). When a target entity in a one-to-one relationship is removed from the relationship, you should cascade the remove operation to the target entity. Such target entities are considered “orphans,” and the orphanRemoval attribute can be used to specify that orphaned entities should be removed.

```
@Entity
@Table(name=”CUSTOMER”)
public class Customer {
private Integer id ;
private String name;
private Cart cart ;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
```

```
@Column(name=”id”,length=20)
public Integer getId() {
return id;

                                        }

public void setId(Integer id) {
this.id = id;

                                        }

@Column(name=”name”,length=50)
public String getName() {
return name;

                                        }

public void setName(String name) {
this.name = name;

                                        }

@OneToOne(cascade=CascadeType.ALL, orphanRemoval=true)
@JoinColumn(name=”CART_ID”)
public Cart getCart() {
return cart;

                                        }

public void setCart(Cart cart) {
this.cart = cart;

                                        }

                                        }
```

The next entity is the Cart containing a @OneToOne reference to Customer using mappedBy attribute, which contains the reference to the variable name used in Customer entity. The mappedBy field owns the relationship and is only specified on the inverse side of the association.

```
@Entity
```

```
@Entity
@Table(name=”CART”)
public class Cart {
private Integer id;
private Double amount;
private Customer customer ;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”id”,length=20)
public Integer getId() {
return id;

                                        }

public void setId(Integer id) {
this.id = id;

                                        }

@Column(name=”amount”,length=20)
public Double getAmount() {
return amount;

                                        }

public void setAmount(Double amount) {
this.amount = amount;

                                        }

@OneToOne(mappedBy=”cart”)
public Customer getCustomer() {
return customer;

                                        }

public void setCustomer(Customer customer) {
this.customer = customer;

                                        }

                                        }
```

}

Table structure

```
DROP TABLE `customer`;
DROP TABLE `cart`;
CREATE TABLE `cart` (
  `id` int(11) NOT NULL AUTO_INCREMENT, `amount` double DEFAULT
NULL,
  PRIMARY KEY (`id`)

);

CREATE TABLE `customer` (
  `id` int(11) NOT NULL AUTO_INCREMENT, `name` varchar(50)
DEFAULT NULL,
  `CART_ID` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY (`CART_ID`),
  CONSTRAINT FOREIGN KEY (`CART_ID`) REFERENCES `cart` (`id`) );
```

Develop the data access tier

Let’s move to the data access tier, beginning with the test case for the get all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class CustomerDaoImplTest {
@Autowired
private CustomerDao dao ;
@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size());

}

}
```

We will see how to write the imitating data access operation for the test overhead. The Hibernate Session.createQuery operation is used with a SQL to

retrieve all the records from the Customer entity.

```
@Repository
@Transactional
public class CustomerDaoImpl implements CustomerDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public List<Customer> getAll() {
return sessionFactory.getCurrentSession().
createQuery("select customer from Customer customer order by customer.id
desc").list(); }

                                                    }
```

Let's now look at the test for insert operation. Insert two rows into the Customer and Cart table and check if the results match 2 records.

```
public class CustomerDaoImplTest {
@Autowired
private CustomerDao dao ;
@Test
public void testInsert(){
Customer customer1 = new Customer();
customer1.setName("Alex");
Cart cart1 = new Cart();
cart1.setAmount(300.0);
customer1.setCart(cart1);
dao.insert(customer1);


Customer customer2 = new Customer();
customer2.setName("Fred");
Cart cart2 = new Cart();
cart2.setAmount(500.0);
customer2.setCart(cart2);
dao.insert(customer2);


Assert.assertEquals(2L, dao.getAll().size());
```

}

}

The conforming insert operation calls the Hibernate Session.save method. The Cart entity identifier gets formed routinely along with the identifier of the Customer entity.

```
public class CustomerDaoImpl implements CustomerDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void insert(Customer customer) {
sessionFactory.getCurrentSession().save(customer); }
}
```

It is justifiable to write the test case using get all records operation to retrieve all the rows and find the relevant record from the list. The test case should not contain hard coded identifier used with get a record using identifier operation. Here a new row is inserted, all the records are retrieved and the matching row is recovered using the identifier of the first row.

```
public class CustomerDaoImplTest {
@Autowired
private CustomerDao dao ;
@Test
public void testGetById() {
Customer customer1 = new Customer();
customer1.setName(“Alex”);
Cart cart1 = new Cart();
cart1.setAmount(300.0);
customer1.setCart(cart1);
dao.insert(customer1);


List<Customer> customers = dao.getAll(); Customer tempCustomer =
customers.get(0);
Customer cust = dao.getById(tempCustomer.getId());
Assert.assertEquals(tempCustomer.getName(), cust.getName()); }

}
```

The equivalent data access method using Hibernate 4 Session.get () procedure is revealed in the following code.

```
public class CustomerDaoImpl implements CustomerDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public Customer getById(int id) {
return (Customer) sessionFactory.getCurrentSession().get(Customer.class,id); }

}
```

We will review the delete test operation now. First, we insert two records followed by a delete and then check whether a solitary record subsists in the database.

```
public class CustomerDaoImplTest {
@Autowired
private CustomerDao dao ;
@Test
public void testDelete() {
Customer customer1 = new Customer();
customer1.setName(“Alex”);
Cart cart1 = new Cart();
cart1.setAmount(300.0);
customer1.setCart(cart1);
dao.insert(customer1);


Customer customer2 = new Customer();
customer2.setName(“Fred”);
Cart cart2 = new Cart();
cart2.setAmount(500.0);
customer2.setCart(cart2);
dao.insert(customer2);


Assert.assertEquals(2L, dao.getAll().size());


dao.delete(customer2);
```

dao.delete(customer2);
Assert.assertEquals(1L, dao.getAll().size());

}

}

The delete operation uses the Hibernate 4 Session.delete () method shown below.

```
public class CustomerDaoImpl implements CustomerDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void delete(Customer customer) {
sessionFactory.getCurrentSession().delete(customer); }
```

}

Lastly, look at the update test method that inserts a row, updates the row, gets all the records and tests whether the fields match.

```
public class CustomerDaoImplTest {
@Autowired
private CustomerDao dao ;
@Test
public void testUpdate() {
Customer customer1 = new Customer();
customer1.setName(“Alex”);
Cart cart1 = new Cart();
cart1.setAmount(300.0);
customer1.setCart(cart1);
dao.insert(customer1);
Assert.assertEquals(1L, dao.getAll().size());
```

List<Customer> customers = dao.getAll(); Customer customer2 = customers.get(0);
customer2.setName(“Fred”);
dao.update(customer2);

List<Customer> customers1 = dao.getAll(); Customer customer3 = customers1.get(0);
Assert.assertEquals(“Fred”, customer3.getName()); }

}

The update data access operation is shown below which uses Hibernate 4 Session.merge () operation.

```
public class CustomerDaoImpl implements CustomerDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void update(Customer customer) {
sessionFactory.getCurrentSession().merge(customer); }

}
```

Develop the business service tier

To develop the business service tier, first look at the test for fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class CustomerServiceImplTest {
@Autowired
private CustomerService service ;
@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size()); }

}
```

Code the identical method in the business service by retrieving all the records using data access operation and convert the entity to data transfer objects using the mapper.

```
@Service
@Transactional
public class CustomerServiceImpl implements CustomerService{
```

```
@Autowired
private CustomerDao dao ;
@Autowired
private CustomerMapper mapper ;
@Override
public List<CustomerDto> findAll() {
List<CustomerDto> customerDtos = new ArrayList<CustomerDto>();
List<Customer> customers = dao.getAll(); for(Customer customer:customers){
customerDtos.add(mapper.mapEntityToDto(customer)); }
return customerDtos;

                                                    }

                                                    }
```

The business service tier uses data access objects, but the data access tier uses entities. The mapper typically has two significant operations which transforms an entity to a data transfer object and vice versa. Note that the Cart identifier is generated every time the Customer is restructured.

```
@Component
public class CustomerMapper {
public Customer mapDtoToEntity(CustomerDto customerDto){
Customer customer = new Customer();
Cart cart = new Cart();
if(null!=customerDto.getId())customer.setId(customerDto.getId());
if(null!=customerDto.getName())customer.setName(customerDto.getName());
if(customerDto.getAmount()>0){
cart.setAmount(customerDto.getAmount());
cart.setCustomer(customer);
customer.setCart(cart);

                                                    }

return customer;

                                                    }

public CustomerDto mapEntityToDto(Customer customer){
CustomerDto customerDto = new CustomerDto() ;
if(null!=customer.getId())customerDto.setId(customer.getId());
```

```
if(null!=customer.getName())customerDto.setName(customer.getName());
if(null!=customer.getCart()){
if(0!=customer.getCart().getAmount())
customerDto.setAmount(customer.getCart().getAmount()); }
return customerDto;

                                                    }

                                                    }
```

The following create test operation creates two occurrences of the entity and checks if the count is double with the fetch all records operation.

```
public class CustomerServiceImplTest {
@Autowired
private CustomerService service ;
@Test
public void testCreate() {
CustomerDto customerDto1 = new CustomerDto();
customerDto1.setName(“Alex”);
customerDto1.setAmount(200);
service.create(customerDto1);


CustomerDto customerDto2 = new CustomerDto();
customerDto2.setName(“Fred”);
customerDto2.setAmount(400);
service.create(customerDto2);


Assert.assertEquals(2L, service.findAll().size()); }

                                                    }
```

The compliant method in the business service tier receives a data access object, translates to the equivalent entity and delegates to the data access tier for saving to the database.

```
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerDao dao ;
@Autowired
```

```
@Autowired
private CustomerMapper mapper ;
@Override
public void create(CustomerDto customerDto) {
dao.insert(mapper.mapDtoToEntity(customerDto)); }

                                                    }
```

The next test operation in line is finding a row with a given identifier.

```
public class CustomerServiceImplTest {
@Autowired
private CustomerService service ;
@Test
public void testFindById() {
CustomerDto customerDto1 = new CustomerDto();
customerDto1.setName(“Alex”);
customerDto1.setAmount(200);
service.create(customerDto1);


List<CustomerDto> customerDtos = service.findAll(); CustomerDto
customerDto = customerDtos.get(0);
CustomerDto customerDto2 = service.findById(customerDto.getId());
Assert.assertEquals(“Alex”, customerDto2.getName()); }

                                                    }
```

The business service operation to find a row with a given identifier is shown below. The data access tier is called to fetch the matching row and the mapper is used to convert to a data transfer object.

```
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerDao dao ;
@Autowired
private CustomerMapper mapper ;
@Override
public CustomerDto findById(int id) {
Customer customer = dao.getById(id);
if(null !=customer){
return mapper.mapEntityToDto(customer);
```

}

return null;

}

}

The remove test operation is next which accepts a particular identifier and removes the corresponding row from the database. The test method creates two samples, removes one sample and checks whether there is one result in the catalogue.

```
public class CustomerServiceImplTest {
@Autowired
private CustomerService service ;
@Test
public void testRemove() {
CustomerDto customerDto1 = new CustomerDto();
customerDto1.setName(“Alex”);
customerDto1.setAmount(200);
service.create(customerDto1);


CustomerDto customerDto2 = new CustomerDto();
customerDto2.setName(“Fred”);
customerDto2.setAmount(400);
service.create(customerDto2);


Assert.assertEquals(2L, service.findAll().size());
List<CustomerDto> customerDtos = service.findAll();
CustomerDto customerDto = customerDtos.get(1);
service.remove(customerDto.getId());
Assert.assertEquals(1L, service.findAll().size()); }
```

}

The comparable business service method is shown in the following code, which

accepts an identifier and removes the identical instance from the databank through the data access tier operation.

```
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerDao dao ;
@Autowired
private CustomerMapper mapper ;
@Override
public void remove(int id) {
dao.delete(dao.getById(id));

                                                    }

}
```

The final test operation is the edit, which creates an instance, edits the name and then checks if the edit was operative.

```
public class CustomerServiceImplTest {
@Autowired
private CustomerService service ;
@Test
public void testEdit() {
CustomerDto customerDto1 = new CustomerDto();
customerDto1.setName(“Alex”);
customerDto1.setAmount(200);
service.create(customerDto1);



customerDto1.setName(“Rose”);
service.edit(customerDto1);



List<CustomerDto> customerDtos = service.findAll(); CustomerDto
customerDto2 = customerDtos.get(0); Assert.assertEquals(“Rose”,
customerDto2.getName()); }

                                                    }
```

The matching operation of the business service is the edit operation which internally calls the data access update method after translation by the mapper.

Note that the edit operation does not have the identifier of the Cart entity and hence generates a new instance.

```
public class CustomerServiceImpl implements CustomerService{
@Autowired
private CustomerDao dao ;
@Autowired
private CustomerMapper mapper ;
@Override
public void edit(CustomerDto customerDto) {
dao.update(mapper.mapDtoToEntity(customerDto)); }

}
```

Develop the presentation tier

The concluding tier in the heap is the REST-based Spring Controller for which we will look at the test first. The test creates an instance of CustomerDto followed by a retrieval and edit of the same instance. Lastly the instance is deleted.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class CustomerControllerTest {
private final Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
hh:mm:ss”).create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;
@Before
public void setUp() {
mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build(); }
@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();
```

```
}

public void testCreate() throws Exception {
CustomerDto customerDto1 = new CustomerDto();
customerDto1.setName(“Alex”);
customerDto1.setAmount(200.0);
String json = gson.toJson(customerDto1);
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“onetoonebidirectionalcreate”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}
public void testUpdate() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get(“onetoonebidirectionalfindAll”); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<CustomerDto>>() {}.getType(); List<CustomerDto>
customerDtos = gson.fromJson(response2, listType); CustomerDto customerDto
= customerDtos.get(0); customerDto.setAmount(400.0);
String json2 = gson.toJson(customerDto);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post(“onetoonebidirectionaledit”);
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}
public void testDelete() throws Exception {
MockHttpServletRequestBuilder requestBuilder =
MockMvcRequestBuilders.get(“onetoonebidirectionalfindAll”); MvcResult
result = this.mockMvc.perform(requestBuilder).andReturn(); String response2 =
result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<CustomerDto>>() {}.getType(); List<CustomerDto>
customerDtos = gson.fromJson(response2, listType); CustomerDto customerDto
= customerDtos.get(0); MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.post(“onetoonebidirectionalremove/”+customerDto.g
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
```

}

}

The complementary REST, Spring Controller, will be communicating with the business service tier which is eventually through the data access tier to connect to the database. The code is like the mock service, with the variance that the code is striking the actual database and not the in memory catalogue.

```
@Controller
@RequestMapping(value=”/onetoonebidirectional”)
@Transactional
public class CustomerController {
@Autowired
private CustomerService service ;
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<CustomerDto> findAll(){
return service.findAll();
```

}

```
@RequestMapping(value=”/findById/{customerid}”,
method=RequestMethod.GET) public @ResponseBody CustomerDto
findById(@PathVariable(“customerid”) Integer customerid){
return service.findById(customerid);
```

}

```
@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void creat(@RequestBody CustomerDto customerDto){
service.create(customerDto);
```

}

```
@RequestMapping(value=”/remove/{customerid}”,
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“customerid”) Integer customerid){
service.remove(customerid);
```

}

```
}

@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody CustomerDto customerDto){
service.edit(customerDto);

}

}
```

We will talk about how we can integrate the REST service with the actual user interface. Please note that the request mapping changes from “*onetoonebidirectional*mock” to “/onetoonebidirectional”. The rest involves writing the actual user interface using the mock user interface as a reference.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a one-to-one bidirectional association
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 5. One-to-One SelfReferencing Relationship

Relationship A one-to-one selfreferencing relationship is a one-to-one relationship which has a reference to its own instance. We will display the familiarized stream of starting with the user interface followed by the service development, then join both units at the close of the chapter. Once you finish reading this chapter you will be able to fully execute a one-to-one selfreferencing relationship, both from a user interface as well as a REST service point of view.

Domain Model

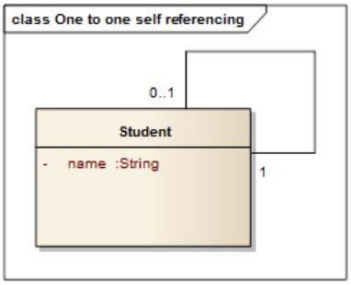

Figure 5-1: A one-to-one selfreferencing relationship We have so far looked at how to manage one-to-one unidirectional and bidirectional relationships. Let’s

start by looking at the one-to-one selfreferencing association shown in Figure 5-1. Here, the Student entity has an identifier and a name field of String type. The Student can have an optional mentor which is also a Student. Hence, within one instance of Student, another instance of Student (who is a mentor) can be associated. We will go with the typical stream of script with the mock user interface initially and then the service.

Develop a User Interface

The three major tasks we look at during the development of the mock user interface are the data transfer object, the mock service and the mock user interface. We will look at them one by one.

Develop the data transfer object

The data transfer object does not have the identifier of the Student mentor object because the internal design of the one-to-one selfreferencing relationship is not exposed to the user interface team. In an instance of a Student object there could be only one mentor object who is also a Student.

```
public class StudentDto {
private Integer id;
private String name;
private String mentorName;

    // getters and setters

                                        }
```

Develop the mock service

The mock service will be interacting with an in-memory custom database for saving the data, which will be consumed by the mock user interface. We will begin with the add feature. We have a list of StudentDto, and with every new entry, one instance of StudentDto having a Student and a mentor is added to the list. Enum oriented Singleton design pattern is used for the in-memory database implementation. The identifier field begins with 1 and increments by 1 for every new count.

```
public enum StudentInMemoryDB {
```

```
INSTANCE;
private static Integer lastId = 0;
private static List<StudentDto> list = new ArrayList<StudentDto>();
public Integer getId() {return ++lastId;}


public void add(StudentDto studentDto){
studentDto.setId(getId());
list.add(studentDto);

                                                    }

                                                    }
```

Now let's look at the equivalent mock service implementation. The create method receives an instance of StudentDto from the user interface and enhances the list via the mock database.

```
@Controller
@RequestMapping(value="onetooneselfreferencemock") public class
StudentMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody StudentDto student){
StudentInMemoryDB.INSTANCE.add(student);

                                                    }

                                                    }
```

Next, look at the find all rows and find with identifier piece.

```
public enum StudentInMemoryDB {
public List<StudentDto> findAll(){ return list; }
public StudentDto findById(Integer id) {
for(StudentDto studentDto : list){
if(studentDto.getId() == id) return studentDto; }
return null ;

                                                    }
```

}

}

In the mock service, the find all rows yields a list of StudentDto and find with identifier gets an instance of StudentDto.

```
@Controller
@RequestMapping(value="onetooneselfreferencemock") public class
StudentMockController {
@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody List<StudentDto>  findAll(){
return StudentInMemoryDB.INSTANCE.findAll();
```

}

```
@RequestMapping(value="/findById/{stuid}", method=RequestMethod.GET)
public @ResponseBody StudentDto findById(@PathVariable("stuid") Integer
stuid){
return StudentInMemoryDB.INSTANCE.findById(stuid); }
```

}

Now instrument the remove operation, which removes a specific row with a specified identifier.

```
public enum StudentInMemoryDB {
public void remove(Integer id) {
StudentDto dto = null;
for(StudentDto studentDto : list){
if(studentDto.getId() ==  id) {
dto = studentDto;
break;
```

}

}

```
if(null != dto) list.remove(dto);
```

}

}

In the alike mock service, the eliminate operation removes a specific row with a certain identifier delivered as a response to the URL.

```
@Controller
@RequestMapping(value=”onetooneselfreferencemock”) public class StudentMockController {
@RequestMapping(value=”/remove/{stuid}”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT) public void remove(@PathVariable(“stuid”) Integer stuid){
StudentInMemoryDB.INSTANCE.remove(stuid);

}

}
```

The edit operation used for changing contents of a row is shown below.

```
public enum StudentInMemoryDB {
public void edit(StudentDto studentDto){
for(StudentDto dto : list){
if(studentDto.getId() == dto.getId()){
dto.setName(studentDto.getName());
dto.setMentorName(studentDto.getMentorName());
break;

}

}

}

}
```

The edit operation receives an amended copy of the StudentDto and appraises the specific row with the variations.

```
@Controller
@RequestMapping(value=”onetooneselfreferencemock”) public class StudentMockController {
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody StudentDto student){
```

StudentInMemoryDB.INSTANCE.edit(student);

}

}

Develop the mock user interface

We saw in the earlier section how to develop the mock service with an in-memory database. Now we will look at the user interface that will call the mock service as shown in Figure 5-2. The user interface will have methods to add, edit, delete and display all records related to the Student, including the mentor.

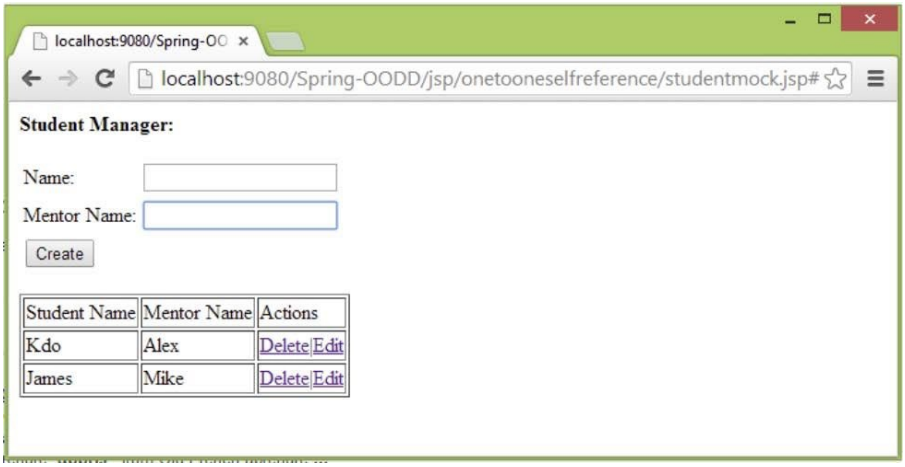

Figure 5-2: User Interface to Manage Students The JSP body is the initial page that loads, after which the JavaScript procedures get called. We will outline the flow with the JSP followed by the Javascript functions.

```
<body onload=”loadObjects()”>
<div id=”container”>
<div>
<p>
<b>Student Manager:</b>
</p>
</div>
<form method=”post” id=”studentForm”>
<table>
<tbody>
<tr>
```

<td>Name:</td>
<td><input name=”stuid” id=”stuid” type=”hidden”> <input name=”stuname” id=”stuname” type=”text”></td> </tr>
<tr>
<td class=”tdLabel”><label>Mentor Name:</label></td> <td><input type=”text” name=”mentor” value=”” id=”mentor” ><td> </tr>
<tr>
<td colspan=”2”><input type=submit value=”Create” id=”subButton” onclick=”return methodCall()”></td> </tr>
</tbody>
</table>
</form>

</body>

First, we will look at the JavaScript loadObjects method, which is called when the page is initialized with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDonetooneselfreference/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)
                                        {
processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“stuname”).value=””; alert(“Error Status Load Objects:”+textStatus); }
                                        });
return false;
                                        }
```

The processResponseData JavaScript procedure called when the above service

call is successful is shown below. The generateTableData JavaScript method, which forms the table structure, is also displayed.

```
function processResponseData(responsedata){
var dynamicTableRow="<table border=1>"+
"<tr>" +
"<td>Student Name</td>"+"<td>Mentor Name</td>"+
"<td>Actions</td>"+
"</tr>";
var dataRow="";
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue,itemno); });
dynamicTableRow=dynamicTableRow+dataRow+"</table>";
document.getElementById("studentFormResponse").innerHTML=dynamicTable
}
function generateTableData(itemvalue,itemNo){
var mentorName = itemvalue.mentorName;
if(null == mentorName) mentorName="";
var dataRow="";
if(mentorName!=""){
dataRow="<tr>" +
"<td>" +itemvalue.name+"</td>"+
"<td>"+mentorName+"</td>"+
"<td>" +
"<a href=# onclick=deleteObject("+itemvalue.id+")>Delete</a>"+
"|<a href=# onclick=editObject("+itemvalue.id+")>Edit</a>"+
"</td>"+
"</tr>";

                                                    }

return dataRow;

                                                    }
```

We have viewed the page loading code. Once you understand how the JSP fragment works, you will know that the JavaScript method methodCall is being called, which is applicable for both create and update functionality.

```
function methodCall(){
var buttonValue = document.getElementById("subButton").value;
if(buttonValue=="Create"){
```

```
if(buttonValue == "Create"){
create();
}else if(buttonValue==”Update”){
update();

                                                    }

return false;

                                                    }
```

Next we will make a note of the create Javascript method that gathers the contents from the text boxes student name and mentor and passes the JSON Object as a StudentDto without an identifier.

```
function create(){
var name = $(“#stuname”).val();
var mentor=$(“#mentor”).val();
var formData={“name”:name,”mentorName”:mentor}; $.ajax({
url : “Spring-OODDonetooneselfreference/mock/create”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

                                                    {

document.getElementById(“stuname”).value=””;
document.getElementById(“mentor”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“stuname”).value=””;
document.getElementById(“mentor”).value=””;
alert(“Error Status Create:”+textStatus);

                                                    }

                                                    });

return false;
```

}

Next we will look at the update JavaScript method.

```
function update(){
var name = $(“#stuname”).val();
var id = $(“#stuid”).val();
var mentor = $(“#mentor”).val();
var formData={“id”:id,”name”:name,”mentorName”:mentor}; $.ajax({
url : “Spring-OODDonetooneselfreference/mock/edit”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
{
document.getElementById(“stuname”).value=””;
document.getElementById(“mentor”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“stuname”).value=””;
document.getElementById(“mentor”).value=””;
alert(“Error Status Update:”+textStatus);
}
});
return false;
}
```

The editObject and the viewObject JavaScript method call allows the selection of a record for edit operation.

```
function editObject(stuid){
editurl=”Spring-OODDonetooneselfreference/mock/findById/”+stuid; var
studentForm={id:stuid};
```

```
$.ajax({
url : editurl,
type: “GET”,
data : studentForm,
dataType: “json”,
success: function(data, textStatus, jqXHR)

                                                {

viewObject(data);
document.getElementById(“subButton”).value=”Update”;
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Find Object:”+textStatus); }

                                                });

                                                 }

function viewObject(data){
document.getElementById(“stuname”).value=data.name;
document.getElementById(“mentor”).value=data.mentorName;
document.getElementById(“stuid”).value=data.id; }
```
Finally, review the delete operation where the student identifier passed as an input and the related Student information is removed.
```
function deleteObject(stuid){
var studentForm={id:stuid};
delurl=”Spring-OODDonetooneselfreference/mock/remove/”+stuid; $.ajax({
url : delurl,
type: “POST”,
data : studentForm,
dataType: “json”,
success: function(data, textStatus, jqXHR)

                                                {

loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus);
```

}

});

}

Develop the Service

The next step is the actual service development, which involves four major steps – creating the entity, creating the data access tier, creating the business service tier, and creating the JSON-based REST service. We will begin with the entity and go bottom up to the REST service.

Develop the resource (entity) tier As a practice, we are following the test driven development approach, hence we will write the test for the entity first and then the entity implementation. First, make two instances of Student with instances of Mentor attached to each Student and then eventually check for four instances of Student, as shown in the following code. Check whether the instance names can be updated as well as deleted.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class StudentTest {
@Autowired
private SessionFactory sessionFactory;
@SuppressWarnings({ “unchecked”, “deprecation” }) @Test
public void testCRUD()

{

Student student1 = new Student();
student1.setName(“Alex”);
```

```
Student mentor1 = new Student();
mentor1.setName(“Fred”);
student1.setMentor(mentor1);
sessionFactory.getCurrentSession().save(student1);
Student student2 = new Student();
student2.setName(“Michel”);
Student mentor2 = new Student();
mentor2.setName(“Mac”);
student2.setMentor(mentor2);
sessionFactory.getCurrentSession().save(student2); List<Student> students =
sessionFactory.getCurrentSession() .createQuery(“select student from Student
student”).list(); Assert.assertEquals(4L, students.size());

Student tempStudent = students.get(1);
tempStudent.setName(“Alex James”);
Student tempmentor = tempStudent.getMentor(); tempmentor.setName(“Fred
James”);
sessionFactory.getCurrentSession().merge(tempStudent); students =
sessionFactory.getCurrentSession() .createQuery(“select student from Student
student”).list(); Assert.assertEquals(4L, students.size());

tempStudent = students.get(3);
sessionFactory.getCurrentSession().delete(tempStudent); students =
sessionFactory.getCurrentSession() .createQuery(“select student from Student
student”).list(); Assert.assertEquals(2L, students.size());

                                                    }

                                                    }
```

Since we are developing a one-to-one selfreferencing relationship, an instance of Student may have an optional instance of Student itself who is a mentor to the Student. We also use orphanRemoval=true because if another instance of mentor is assigned to the Student, the old mentor will be deleted. Note the use of nullable attribute as true in order to make the mentor optional, since the mentor is also a Student and the mentor’s mentor is not required.

```
@Entity
@Table(name=”STUDENT”)
public class Student {
private Integer id;
```

```
private Integer id;
private String name;
private Student mentor;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”,length=20)
public Integer getId() {
return id;

                                                    }

public void setId(int id) {
this.id = id;

                                                    }

@Column(name=”name”,length=50)
public String getName() {
return name;

                                                    }

public void setName(String name) {
this.name = name;

                                                    }

@OneToOne(cascade=CascadeType.ALL, orphanRemoval=true)
@JoinColumn(name=”mentor_id”,nullable=true)
public Student getMentor() {
return mentor;

                                                    }

public void setMentor(Student mentor) {
this.mentor = mentor;

                                                    }

                                                    }
```

Table structure

```
DROP TABLE `student`;
CREATE TABLE `student` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `name` varchar(50)
DEFAULT NULL, `mentor_id` int(11) DEFAULT NULL, PRIMARY KEY
(`ID`),
  KEY (`mentor_id`),
  CONSTRAINT FOREIGN KEY (`mentor_id`) REFERENCES `student` (`ID`)
);
```

Develop the data access tier

Now let's turn to the data access tier. We will start with the test case for the get all records procedure.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class StudentDaoImplTest {
@Autowired
private StudentDao dao ;
@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size()); }

                                                    }
```

For the test above, let's look at the corresponding data access operation. The Hibernate Session.createQuery method is used with a SQL to retrieve all the records from the Student entity.

```
@Repository
@Transactional
public class StudentDaoImpl implements StudentDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public List<Student> getAll() {
return (List<Student>)sessionFactory.getCurrentSession().
createQuery("select student from Student student").list(); }
```

createQuery("select student from Student student").list(); }

}

Let's now observe the test for insert operation. We insert a few rows into the Student entity along with his or her mentor and check if the results match the four records in the Student entity.

```
public class StudentDaoImplTest {
@Autowired
private StudentDao dao ;
@Test
public void testInsert(){
Student student1 = new Student();
student1.setName("Alex");
Student mentor1 = new Student();
mentor1.setName("Fred");
student1.setMentor(mentor1);
dao.insert(student1);


Student student2 = new Student();
student2.setName("Michel");
Student mentor2 = new Student();
mentor2.setName("Mac");
student2.setMentor(mentor2);
dao.insert(student2);
Assert.assertEquals(4L, dao.getAll().size()); }

}
```

The insert operation just requests the Hibernate Session.save method. The Student entity identifier is created along with the identifier of the Student mentor.

```
public class StudentDaoImpl implements StudentDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void insert(Student student) {
sessionFactory.getCurrentSession().save(student); }
```

}
Next, note how to find a row with an identifier. If you specify an identifier explicitly, it may work in your system but may fail in another system, hence the code will not be portable.

```
public class StudentDaoImplTest {
@Autowired
private StudentDao dao ;
@Test
public void testGetById() {
Student student1 = new Student();
student1.setName(“Alex”);
Student mentor1 = new Student();
mentor1.setName(“Fred”);
student1.setMentor(mentor1);
dao.insert(student1);
List<Student> students = dao.getAll();
Student student = students.get(1);
Student student2 = dao.getById(student.getId()); Assert.assertEquals(“Alex”,
student2.getName()); Assert.assertEquals(“Fred”,
student2.getMentor().getName()); }
```

}

The data access method that uses Hibernate 4 Session.get () procedure is shown below which accepts the identifier and retrieves the matching Student entity.

```
public class StudentDaoImpl implements StudentDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public Student getById(Integer id) {
return (Student)sessionFactory.getCurrentSession().get(Student.class, id); }
```

}

We will view the delete test procedure at this time. First, we insert four records of Student and mentor followed by a delete of a Student along with its mentor and then check whether a single record lives in the database.

```
public class StudentDaoImplTest {
@Autowired
private StudentDao dao ;
```

```
private StudentDao dao ;
@Test
public void testDelete() {
Student student1 = new Student();
student1.setName(“Alex”);
Student mentor1 = new Student();
mentor1.setName(“Fred”);
student1.setMentor(mentor1);
dao.insert(student1);


Student student2 = new Student();
student2.setName(“Michel”);
Student mentor2 = new Student();
mentor2.setName(“Mac”);
student2.setMentor(mentor2);
dao.insert(student2);
Assert.assertEquals(4L, dao.getAll().size());
List<Student> students = dao.getAll();
Student tempStudent = students.get(3);
dao.delete(tempStudent);
Assert.assertEquals(2L, dao.getAll().size()); }

                                                    }
```

The delete operation uses the Hibernate 4 Session.delete () method mentioned below.

```
public class StudentDaoImpl implements StudentDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void delete(Student student) {
sessionFactory.getCurrentSession().delete(student); }

                                                    }
```

The update test method now inserts a row, updates the row and tests whether the count is the same.

```
public class StudentDaoImplTest {
@Autowired
```

```
private StudentDao dao ;
@Test
public void testUpdate() {
Student student1 = new Student();
student1.setName(“Alex”);
Student mentor1 = new Student();
mentor1.setName(“Fred”);
student1.setMentor(mentor1);
dao.insert(student1);
Assert.assertEquals(2L, dao.getAll().size());
List<Student> students = dao.getAll();
Student tempStudent = students.get(1);
tempStudent.setName(“Alex James”);
Student tempMentor = tempStudent.getMentor(); tempMentor.setName(“Fred
James”);
tempStudent.setMentor(tempMentor);
dao.update(tempStudent);
Assert.assertEquals(2L, dao.getAll().size()); }

                                                    }
```

The update data access action then practices the Hibernate 4 Session.merge () operation.

```
public class StudentDaoImpl implements StudentDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void update(Student student) {
sessionFactory.getCurrentSession().merge(student); }

                                                    }
```

Develop the business service tier

Let’s now jump to the business service tier. First, we will review the test for fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
```

```
public class StudentServiceImplTest {
@Autowired
private StudentService service ;
@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size()); }

                                                    }
```

Let us look at how to code the matching method in the business service. We retrieve all the records using the data access operation and return a list of StudentDto using the mapper.

```
@Service
@Transactional
public class StudentServiceImpl implements StudentService{
@Autowired
private StudentDao dao ;
@Autowired
private StudentMapper mapper ;
@Override
public List<StudentDto> findAll() {
List<StudentDto> studentDTOs = new ArrayList<StudentDto>(); List<Student>
students = dao.getAll();
if(null != students){
for(Student student : students){
studentDTOs.add(mapper.mapEntityToDto(student)); }

                                                    }

return studentDTOs;

                                                    }

                                                    }
```

As discussed earlier, we have a mapper for converting data access objects into entities and vice-versa, which typically has two significant operations – mapDtoToEntity and mapEntityToDto as shown below. Note that the Student mentor identifier is generated every time the Student is created. This is important because the management of the identifier of the mentor should not be delegated

to the user interface. The user interface will pass the name of the mentor and the service will manage the creation and updating of the mentor who is also a Student.

```
@Component
public class StudentMapper {
public Student mapDtoToEntity (StudentDto studentDTO){
Student student = new Student();
Student mentor = new Student();
if(null !=studentDTO.getId() && studentDTO.getId()> 0)
student.setId(studentDTO.getId()); if(null !=studentDTO.getName())
student.setName(studentDTO.getName()); if(null
!=studentDTO.getMentorName())
{mentor.setName(studentDTO.getMentorName());student.setMentor(mentor);}
return student;

                                                    }

public StudentDto mapEntityToDto (Student student){
StudentDto studentDTO = new StudentDto();
if(null !=student.getId()) studentDTO.setId(student.getId()); if(null
!=student.getName()) studentDTO.setName(student.getName()); if(null
!=student.getMentor())studentDTO.setMentorName(student.getMentor().getNam
return studentDTO;

                                                    }

                                                    }
```

The create test operation builds an instance of Student along with its mentor and checks if the fetch all records has two records.

```
public class StudentServiceImplTest {
@Autowired
private StudentService service ;
@Test
public void testCreate() {
StudentDto studentDTO = new StudentDto();
studentDTO.setName(“Alex”);
studentDTO.setMentorName(“Fred”);
service.create(studentDTO);
Assert.assertEquals(2L, service.findAll().size()); }
```

}

The matching method in the business service tier receives a data access object, translates to the equivalent entity and passes it to the data access tier for saving to the database.

```
public class StudentServiceImpl implements StudentService{
@Autowired
private StudentDao dao ;
@Autowired
private StudentMapper mapper ;
@Override
public void create(StudentDto studentDTO) {
dao.insert(mapper.mapDtoToEntity(studentDTO)); }
```

}

The following test operation is finding a row with a given identifier which uses the fetch all records operation to retrieve the identifier of the second student. This identifier is then passed to fetch record with an identifier method and checked as to whether the instance is not null. Note that it is important to retrieve the identifiers using the fetch all records operation and not hard code any identifier resulting in non-portable test code.

```
public class StudentServiceImplTest {
@Autowired
private StudentService service ;
@Test
public void testFindById() {
StudentDto studentDTO = new StudentDto();
studentDTO.setName(“Alex”);
studentDTO.setMentorName(“Fred”);
service.create(studentDTO);
List<StudentDto> studentDTOs = service.findAll(); StudentDto stDto =
studentDTOs.get(1) ;
StudentDto stDto2 = service.findById(stDto.getId());
Assert.assertNotNull(stDto2);
```

}

}

,

The business service operation to find a row with a given identifier is shown below.

```
public class StudentServiceImpl implements StudentService{
@Autowired
private StudentDao dao ;
@Autowired
private StudentMapper mapper ;
@Override
public StudentDto findById(Integer id) {
Student student = dao.getById(id);
if(null !=student){
return mapper.mapEntityToDto(student);

                                                    }

return null;

                                                    }

                                                    }
```

The test operation for remove creates two instances, removes one of them, and checks if only one record persists in the database. This is to check whether the remove operation actually eliminates the instance.

```
public class StudentServiceImplTest {
@Autowired
private StudentService service ;
@Test
public void testRemove() {
StudentDto studentDTO = new StudentDto();
studentDTO.setName(“Alex”);
studentDTO.setMentorName(“Fred”);
service.create(studentDTO);



StudentDto stuDto = new StudentDto();
stuDto.setName(“Smith”);
stuDto.setMentorName(“Flex”);
```

```
service.create(stuDto);
Assert.assertEquals(4L, service.findAll().size());
List<StudentDto> studentDTOs = service.findAll(); StudentDto stDto =
studentDTOs.get(3) ;
service.remove(stDto.getId());
Assert.assertEquals(2L, service.findAll().size());

                                                    }

                                                    }
```

The business service method is defined, which retrieves an instance by an identifier and removes the same from the catalogue via the data access tier procedure.

```
public class StudentServiceImpl implements StudentService{
@Autowired
private StudentDao dao ;
@Autowired
private StudentMapper mapper ;
@Override
public void remove(Integer id) {
Student student = dao.getById(id);
if(null != student)
dao.delete(student);

                                                    }

}
```

The next test operation is the edit, which creates an instance, edits the student name and mentor name, and then checks if the edit was functioning.

```
public class StudentServiceImplTest {
@Autowired
private StudentService service ;
@Test
public void testEdit() {
StudentDto studentDTO = new StudentDto();
studentDTO.setName(“Alex”);
studentDTO.setMentorName(“Fred”);
service.create(studentDTO);
List<StudentDto> studentDTOs = service.findAll(); StudentDto sDto =
```

```
studentDTOs.get(1);
sDto.setName(“James Alex”);
sDto.setMentorName(“Fred James”);
service.edit(sDto);
List<StudentDto> students = service.findAll(); Assert.assertEquals(2L,
students.size());

                                                    }

                                                    }
```

The equivalent operation of the business service is the edit operation, which bypasses the call to the data access, the update method after transformation with the mapper method. Note that the edit operation does not have the identifier of the Student mentor entity and hence generates a new one.

```
public class StudentServiceImpl implements StudentService{
@Autowired
private StudentDao dao ;
@Autowired
private StudentMapper mapper ;
@Override
public void edit(StudentDto studentDTO) {
dao.update(mapper.mapDtoToEntity(studentDTO)); }

                                                    }
```

Develop the presentation tier

The last tier in the stack is the REST-based Spring Controller. Here we will first describe the test. The test first creates an instance of StudentDto using the REST create service. This is followed by a retrieval of all the records and an update of the only instance of StudentDto. Finally the test removes the instance by getting the identifier from the fetch all records service.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class StudentControllerTest {
private Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
```

```
hh:mm:ss″).create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;
@Before
public void setUp() {
mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build(); }
@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                                    }

public void testCreate() throws Exception {
StudentDto studentDto = new StudentDto();
studentDto.setName(“Alex”);
studentDto.setMentorName(“Fred”);
String json = gson.toJson(studentDto);
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“onetooneselfreferencecreate”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}
public void testUpdate() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get(“onetooneselfreferencefindAll”); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<StudentDto>>() {}.getType(); List<StudentDto> studentDtos
= gson.fromJson(response2, listType); StudentDto studentDto =
studentDtos.get(0);
studentDto.setName(“Alex”);
studentDto.setMentorName(“Fred James”);
String json2 = gson.toJson(studentDto);
MockHttpServletRequestBuilder requestBuilder3 =
```

MockMvcRequestBuilders.post(“*onetooneselfreference*edit”);
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st

}

public void testDelete() throws Exception {
MockHttpServletRequestBuilder requestBuilder =
MockMvcRequestBuilders.get(“*onetooneselfreference*findAll”); MvcResult
result = this.mockMvc.perform(requestBuilder).andReturn(); String response2 =
result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<StudentDto>>() {}.getType(); List<StudentDto> studentDtos
= gson.fromJson(response2, listType); StudentDto studentDto2 =
studentDtos.get(0);
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.post(“*onetooneselfreference*remove/”+studentDto2.ge
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
}

}

The resulting REST, Spring Controller will be connecting with the business service tier which is ultimately through the data access tier to link to the database. The program is similar to the mock service, with the distinction that the code here is interacting with the actual database and not the in memory log.
@Controller
@RequestMapping(value=”/onetooneselfreference”) public class
StudentController {
@Autowired
private StudentService service ;
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<StudentDto> findAll(){
return service.findAll();

}

@RequestMapping(value=”/findById/{stuid}”, method=RequestMethod.GET)
public @ResponseBody StudentDto findById(@PathVariable(“stuid”) Integer

```
stuid){
return service.findById(stuid);

                                                    }

@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody StudentDto student){
service.create(student);

                                                    }

@RequestMapping(value="/remove/{stuid}", method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT) public void
remove(@PathVariable("stuid") Integer stuid){
service.remove(stuid);

                                                    }

@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody StudentDto student){
service.edit(student);

                                                    }

                                                    }
```

The ultimate step in the list is the integration of the REST service with the genuine user interface. Note that the request mapping changes from "*onetooneselfreference*mock" to "/onetooneselfreference". The remaining comprises writing the real user interface using the mock user interface as a reference.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database

- Develop mock user interface
- Develop entity related to a one-to-one selfreferencing association
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Part III. Managing One-to-Many Relationships

Chapter 6. One-to-Many Unidirectional Relationship

Relationship A one-to-many unidirectional relationship is a one-to-many relationship which has reference from parent to child but not vice versa. We will follow the customary flow of starting with the user interface followed by the service development and integrating both the modules at the end of the chapter. Following the chapter closely will enable you to master the user interface as well as a REST service related to a one-to-many unidirectional relationship.

Domain Model

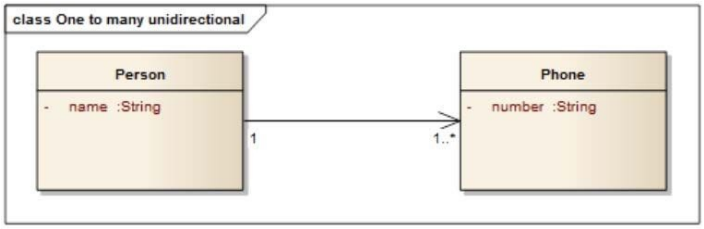

Figure 6-1: A one-to-many unidirectional relationship We have seen how to manage the different forms of one-to-one relationships in the previous chapters. In this chapter, we will see a one-to-many unidirectional association, as shown in Figure 6-1. The Person entity has an identifier and a name field of type String.

The Phone entity also has an identifier and a number String field. In an instance of Person, there must be one or more instances of Phone associated. Similarly, in an instance of Phone there must be one instance of Person linked. Since the relationship is unidirectional, you can traverse the list of Phones from Person but the opposite traversal is not allowed. We will follow the pattern of writing the mock user interface initially and then the genuine service.

Develop a User Interface

The three key tasks we look at during the growth of the mock user interface are the creation of the data transfer object, the mock service and the mock user interface. We will proceed one by one in order.

Develop the data transfer object

The Person data transfer object has an identifier, the name of the person and a list of String numbers. The identifier of the Phone object is hidden from the caller, although an instance of Person can have one or more instances of Phone numbers. The idea is not to expose the database design because of the complexity and to make it simple for the caller to understand the API.

```
public class PersonDto {
private Integer id;
private String name;
private List<String> numbers;

// getters and setters

}
```

Develop the mock service

The mock service will network with an in-memory convention database for saving the data which will be consumed by the mock user interface. We will initially see the add functionality. We have a list of PersonDto and with every new instance, one instance of PersonDto having a Person and phone numbers is added to the list. A Singleton design pattern using enum is followed for the in-memory database execution. The identifier field starts with 1 and advances by 1

for every new entry.
public enum PersonInMemoryDB {
INSTANCE;
private static Integer lastId = 0;
private static List<PersonDto> list = new ArrayList<PersonDto>(); public
Integer getId() {
return ++lastId;

}

public void add(PersonDto personDto){
personDto.setId(getId());
list.add(personDto);

}

}

Now look at the mock service implementation. The create method collects an instance of PersonDto from the user interface and appends the list via the mock database.
@Controller
@RequestMapping(value="*onetomanyunidirectional*mock") public class
PersonMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody PersonDto personDto){
PersonInMemoryDB.INSTANCE.add(personDto);

}

}

Next, we will proceed with the find all rows and find with identifier section.
public enum PersonInMemoryDB {
public List<PersonDto> findAll(){
return list;

}

public PersonDto findById(Integer id) {

```
public PersonDto findById(Integer id) {
for(PersonDto personDto : list){
if(personDto.getId() == id) return personDto ; }
return null;

                                                    }

                                                    }
```

In the mock service, the find all rows produces a list of PersonDto, and find with identifier contracts an instance of PersonDto. The find all rows returns the list embedded in the Person In-memory database. The find with identifier service matches each instance of PersonDto from the in-memory list and returns the matching object.

```
public class PersonMockController {
@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody List<PersonDto> findAll(){
return PersonInMemoryDB.INSTANCE.findAll();

                                                    }

@RequestMapping(value="/findById/{personid}",
method=RequestMethod.GET) public @ResponseBody PersonDto
findById(@PathVariable("personid") Integer personid){
return PersonInMemoryDB.INSTANCE.findById(personid); }

                                                    }
```

Next, we will implement the remove operation which confiscates a specific row with a listed identifier. The in-memory list is traversed to find the matching PersonDto with the identifier provided. When a match is found, the instance is removed from the in-memory list.

```
public enum PersonInMemoryDB {
public void remove(Integer id) {
PersonDto dto = null;
for(PersonDto personDto : list){
if(personDto.getId() == id){dto = personDto;break;}

                                                    }

if(null !=dto){list.remove(dto);}
```

if(num > dto){instance.remove(dto);}

}

}

In the comparable mock service, the exclude operation removes a specific row with the defined identifier supplied as a parameter to the URL. The identifier is fed to the in-memory database's remove operation.

```
public class PersonMockController {
@RequestMapping(value="/remove/{personid}",
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable("personid") Integer personid){
PersonInMemoryDB.INSTANCE.remove(personid);

}

}
```

Next, look at the edit operation responsible for modernizing a row. The name and the set of numbers is updated with the matching identifier present in PersonDto.

```
public enum PersonInMemoryDB {
public void edit(PersonDto personDto){
for(PersonDto dto : list){
if(dto.getId() == personDto.getId()){
dto.setName(personDto.getName());
dto.setNumbers(personDto.getNumbers());

}

}

}

}
```

The edit operation accepts an altered copy of the PersonDto and updates the specific row with the changes.

```
public class PersonMockController {
@RequestMapping(value="/edit",method=RequestMethod.POST)
```

```
@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody PersonDto personDto){
PersonInMemoryDB.INSTANCE.edit(personDto);

                                                    }

                                                    }
```

Develop the mock user interface

Now that we have seen how to write the mock service with an in-memory database, let's write the user interface that will use this mock service. The user interface will have the means to add, edit, delete and show all records related to the Person, including the phone numbers. A screenshot of the user interface is shown in Figure 6-2.

Figure 6-2: A screenshot of the Person and Phone user interface The JSP body is the opening page which is loaded initially and after that the JavaScript actions gets called. We will start with the JSP fragment and then follow it with the JavaScript functions.

```
<body onload="loadObjects()">
<div id="container">
<div>
<p>
<b>Person Manager:</b>
```

</p>

<form method=”post” id=”personForm”>

<label>Name:<input type=”text” name=”name” id=”name”></label> <label>
<input type=”hidden” name=”personid” id=”personid”></label>

<label>Phone:<input type=”text” name=”phone” id=”phone” size=”10”
maxlength=”12”>(+)
</label> <label id=”removeLink_1”></label>

<input type=submit value=”Create” id=”subButton”
onclick=”return methodCall()”> </form>

</body>
First, look at the JavaScript loadObjects method which is called when the page is prepared with the onload event. The loadObjects function is responsible for drawing the grid which contains the list of Person objects.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDonetomanyunidirectional/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR){
processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown){
document.getElementById(“name”).value=””;
alert(“Error Status Load Objects:”+textStatus); }

                                                    });

return false;

                                                    }
```

The processResponseData JavaScript method is called when the above service call is effective. The generateTableData JavaScript method, which constructs the table grid, is also shown.

```
function processResponseData(objects){
var dyanamicTableRow="<table border=1>"+"<tr>" +"<td>Person
Name</td>"+"<td>Numbers</td>"+"<td>Actions</td>"+"</tr>"; var
dataRow="";
$.each(objects, function(itemno, object)
{dataRow=dataRow+generateTableData(object);});
dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("personFormResponse").innerHTML=dyanamicTabl
}

function generateTableData(object){
var phones = object.numbers;
var dataRow="<tr>" +"<td>" +object.name+"</td>";dataRow =
dataRow+"<td>"+"<select>"+"<option>"+"</option>"; if(phones!=null &&
phones.length > 0){
for(var index=0;index<phones.length;index++){
var phone = phones[index];
dataRow = dataRow+"<option value="+phone+">"+phone+"</option>"; }

                                                    }

dataRow=dataRow+"</select></td>";dataRow=dataRow+"<td>" +"<a href=#
onclick=deleteObject("+object.id+")>Delete</a>"+"|<a href=#
onclick=editObject("+object.id+")>Edit</a>"+"</td>"+"</tr>"; return dataRow;

                                                    }
```

We have seen the page loading technique with the above code. Notice how the JavaScript method methodCall is being called, which is applicable for both create and update functionality. When the button has a value "Create", the create JavaScript function is called and when the value is "Update", the update JavaScript method is called.

```
function methodCall(){
var buttonValue = document.getElementById("subButton").value;
if(buttonValue=="Create"){
create();
}else if(buttonValue=="Update"){
```

```
}else if(buttonValue == "Update"){
update();

                                        }

return false;

                                        }
```

Look at how the create JavaScript method takes the inputs from the person name and phone numbers and constructs a PersonDto JSON Object to be passed as a parameter to the mock create service. The name is retrieved from an input box and the set of phone numbers is pushed into an array from elements having the name “phone”.

```
function create(){
var name = $("#name").val();
var numbers = null;
var flag= false;
var phone=document.getElementsByName("phone");
if(phone.length > 0){
numbers =  new Array();
for(var i=0;i<phone.length;i++){
if(trim(phone[i].value)!=""){
numbers.push(phone[i].value);
flag=true;

                                        }

                                        }

                                        }

if(flag==false){numbers=null;}
var formData={"name":name,"numbers":numbers};
$.ajax({url : "Spring-OODDonetomanyunidirectional/mock/create", type:
"POST",data :
JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR){
```

```
document.getElementById("name").value="";
document.getElementById("subButton").value="Create"; loadObjects();
initiatePhoneBoxGroup();

},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("name").value="";
initiatePhoneBoxGroup();
alert("Error Status Create:"+textStatus);

                                                    }

                                                   });

return false;

                                                    }
```

Now look at the initiatePhoneBoxGroup JavaScript method which resets the phone numbers user interface area.

```
function initiatePhoneBoxGroup(){
var phoneBoxGroupDiv = document.getElementById("phoneBoxGroup");
phoneBoxGroupDiv.innerHTML="";
var phoneDiv = document.createElement('div');
phoneDiv.setAttribute("id","phoneBox_1");
var phoneBox_1 = "<label>Phone:<input type='text' name='phone' id='phone'
size='10' maxlength='12'><a href=# onclick=javascript:addMorePhone()>(+)
</a></label>"; var removeLink_1="<label id='removeLink_1'></label>"; var
setUrl = phoneBox_1+removeLink_1;
phoneDiv.innerHTML=setUrl;
phoneBoxGroupDiv.appendChild(phoneDiv);

                                                    }
```

Next we will review the update JavaScript method which constitutes a PersonDto including an identifier, name and phone numbers. The update retrieves the identifier of the Person from the hidden field, the name from the input box and the set of numbers retrieved from elements having the same name pushed into an array.

```
function update(){
```

```
var name = $(“#name”).val();
var id = +$(“#personid”).val();
var flag= false;
var phone=document.getElementsByName(“phone”);
if(phone.length > 0){
numbers =  new Array();
for(var i=0;i<phone.length;i++){
if(trim(phone[i].value)!=””){
numbers.push(phone[i].value);
flag=true;

                                                    }

                                                    }

                                                    }

if(flag==false){numbers=null;}
var formData={“id”:id,”name”:name,”numbers”:numbers}; $.ajax({
url : “Spring-OODDonetomanyunidirectional/mock/edit”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

                                                    {

document.getElementById(“name”).value=””;
initiatePhoneBoxGroup();
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“name”).value=””;
initiatePhoneBoxGroup();
alert(“Error Status Update:”+textStatus);

                                                    }

                                                    });
```

return false;

}

The editObject and viewObject JavaScript method calls are used to retrieve the selected object and display for editing.

```
function editObject(personid){
editurl=”Spring-OODDonetomanyunidirectional/mock/findById/”+personid; var
personForm={id:personid};
$.ajax({url : editurl,
type: “GET”,
data : personForm,
dataType: “json”,
success: function(data, textStatus, jqXHR){
viewObject(data);
document.getElementById(“subButton”).value=”Update”; },
error: function (jqXHR, textStatus, errorThrown){
alert(“Error Status Find Object:”+textStatus); }

});

}


function viewObject(data){
var index = 1;
document.getElementById(“name”).value=data.name;
document.getElementById(“personid”).value=data.id; var numbers =
data.numbers;
if(numbers!=null){
initiatePhoneBoxGroup();
var phoneBoxGroupDiv = document.getElementById(“phoneBoxGroup”);
for(var cnt=0;cnt<numbers.length;cnt++){
if(cnt==0){
var phones= document.getElementsByName(“phone”);
phones[cnt].value=numbers[cnt];
}else{
index++;
var phoneDiv = document.createElement(‘div’);
phoneDiv.setAttribute(“id”,”phoneBox_”+index); var
```

```
remId=”removeLink_”+index;
var removeLinkUrl = “|<label id=”+remId+”><a href=’#’
onclick=’removePhone(“+index+”)’>(-)</a></label>”; phoneDiv.innerHTML =
“<label>Phone:”+”<input type=’text’ id=’phone’ name=’phone’
mqxlength=’12’ size=’10’ value=’”+numbers[cnt]+”’/>
</label>”+removeLinkUrl; phoneBoxGroupDiv.appendChild(phoneDiv);

                                                    }

                                                    }

                                                    }

                                                    }
```

Finally, review the delete operation where the person identifier passed as a response and the related Person information is removed.

```
function deleteObject(personid){
var personForm={id:personid};
delurl=”Spring-OODDonetomanyunidirectional/mock/remove/”+personid;
$.ajax({url : delurl,
  type: “POST”,
  data : personForm,
  dataType: “json”,
  success: function(data, textStatus, jqXHR){
  loadObjects();
  },
  error: function (jqXHR, textStatus, errorThrown){
  alert(“Error Status Delete:”+textStatus); }

                                                    });

                                                    }
```

Let’s look more closely at the phone numbers area of adding and deleting text boxes. This is done with the addMorePhone and removePhone Javascript method calls and allows multiple numbers to be managed. This is an important feature as it manages the user interface area containing the phone numbers where at least one phone number is required.

```
var counter = 1;
```

```
function addMorePhone(){
counter++;
var phoneDiv = document.createElement('div');
phoneDiv.setAttribute("id","phoneBox"+"_"+counter); var
remId="removeLink_"+counter;
var removeLinkUrl = "|<label id="+remId+"><a href='#'
onclick='removePhone("+counter+")'>(-)</a></label>"; var
phoneBoxGroupDiv = document.getElementById("phoneBoxGroup");
phoneDiv.innerHTML = "<label>Phone:"+"<input type='text' id='phone'
name='phone' mqxlength='12' size='10'/></label>"+removeLinkUrl;
phoneBoxGroupDiv.appendChild(phoneDiv);

                                                    }

function removePhone(index){
var phonelist=document.getElementsByName("phone"); var size =
phonelist.length;
if(size==1){counter=1;return false;}
var phoneBoxGroupDiv = document.getElementById("phoneBoxGroup");
phoneBoxGroupDiv.removeChild(document.getElementById("phoneBox"+"_"+
}
```

Develop the Service

The next step is the real service development, which involves four main steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON-based REST facility. We will go bottom up starting with the entity to the REST service.

Develop the resource (entity) tier First, we create two instances of Person with phone numbers and then eventually check for two instances of Person. Next we do an update and test whether the changes are reflected. Lastly, we remove one occurrence and test whether the count is solitary.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class PersonTest {
@Autowired
private SessionFactory sessionFactory;
@Test
public void testCRUD()

                                                {

Person p1 = new Person();
p1.setName(“Alex”);
List<Phone> phones = new ArrayList<Phone>(); Phone ph1 = new Phone();
ph1.setNumber(“7798989138”);
Phone ph2 = new Phone();
ph2.setNumber(“7798989169”);
phones.add(ph1);
phones.add(ph2);
p1.setPhones(phones);
sessionFactory.getCurrentSession().save(p1);



Person p2 = new Person();
p2.setName(“Fred”);
List<Phone> phones2 = new ArrayList<Phone>(); Phone phone1 = new
Phone();
phone1.setNumber(“8836987”);
phones2.add(phone1);
p2.setPhones(phones2);
sessionFactory.getCurrentSession().save(p2);



List<Person> persons = sessionFactory.getCurrentSession() .createQuery(“select
p from Person p order by id asc”).list(); Assert.assertEquals(2L,persons.size());



Person p = persons.get(0);
p.setName(“Jhonson”);
```

```
p.setName("Jhonson");
sessionFactory.getCurrentSession().update(p);


List<Person> persons2 = sessionFactory.getCurrentSession()
.createQuery("select p from Person p order by id asc").list();
Person tp = persons2.get(0);
Assert.assertEquals("Jhonson", tp.getName());


sessionFactory.getCurrentSession().delete(tp);
List<Person> persons3 = sessionFactory.getCurrentSession()
.createQuery("select p from Person p order by id asc").list();
Assert.assertEquals(1L,persons3.size());

                                                    }

                                                    }
```

We will investigate the Phone entity which contains an identifier and a number String field. Since the entity is on the inverse side and is a unidirectional association, no reference to Person is mentioned.

```
@Entity
@Table(name="PHONE")
public class Phone {
private Integer id;
private String number;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="ID",length=20)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }
```

```
@Column(name=”NUMBER”,length=20)
public String getNumber() {
return number;

                                                    }

public void setNumber(String number) {
this.number = number;

                                                    }

                                                    }
```

Since we are developing a one-to-many unidirectional relationship, an instance of Person must have one or more instances of Phone. Since the relationship is unidirectional, a reference to a list of Phone is stated. We also use orphanRemoval=true because if a set of Phones is reassigned the old Phones will be deleted. The Person is the owner and a reference to the identifier of the Person is referenced via this entity using @JoinColumn annotation.

```
@Entity
@Table(name=”PERSON”)
public class Person {
private Integer id;
private String name;
private List<Phone> phones ;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”,length=20)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }

@Column(name=”name”,length=50)
public String getName() {
```

```
return name;

                                                    }

public void setName(String name) {
this.name = name;

                                                    }

@OneToMany(cascade=CascadeType.ALL,orphanRemoval=true)
@JoinColumn(name="PERSON_ID",referencedColumnName="ID") public
List<Phone> getPhones() {
return phones;

                                                    }

public void setPhones(List<Phone> phones) {
this.phones = phones;

                                                    }

                                                    }
```

Table structure

```
DROP TABLE `phone`;
DROP TABLE `person`;
CREATE TABLE `person` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `name` varchar(50)
DEFAULT NULL,
  PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `phone` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `NUMBER` varchar(20)
DEFAULT NULL, `PERSON_ID` int(11) DEFAULT NULL, PRIMARY KEY
(`ID`),
  KEY (`PERSON_ID`),
  CONSTRAINT FOREIGN KEY (`PERSON_ID`) REFERENCES `person`
```

(`ID`));

Develop the data access tier

Next we will look at the data access tier. We will start with the test case for the get all records formula.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class PersonDaoImplTest {
@Autowired
private PersonDao dao ;
@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size());

}

}
```

Note how to write the imitating data access operation for the test directly above. The Hibernate Session.createQuery method is used with a SQL to retrieve all the records from the Person entity.

```
@Repository
@Transactional
public class PersonDaoImpl implements PersonDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public List<Person> getAll() {
return (List<Person>)sessionFactory.getCurrentSession().
createQuery(“select person from Person person order by person.id desc”).list(); }

}
```

Subsequently, let’s look at the test for insert operation. We insert a single row into the Person entity along with Phone numbers and check if the results match a solitary record.

```
public class PersonDaoImplTest {
@Autowired
private PersonDao dao ;
@Test
public void testInsert(){
Person p1 = new Person();
p1.setName(“Alex”);
List<Phone> phones = new ArrayList<Phone>(); Phone ph1 = new Phone();
ph1.setNumber(“7798989138”);
Phone ph2 = new Phone();
ph2.setNumber(“7798989169”);
phones.add(ph1);
phones.add(ph2);
p1.setPhones(phones);
dao.insert(p1);
Assert.assertEquals(1L, dao.getAll().size());

                                                    }

                                                    }
```

The emulating insert operation just requests the Hibernate Session.save method. The Person entity identifier gets arranged reliably along with the identifier of the Phone.

```
public class PersonDaoImpl implements PersonDao {
@Autowired
private SessionFactory sessionFactory ;

@Override
public void insert(Person person) {
sessionFactory.getCurrentSession().save(person); }
}
```

We will see afterward how to put together an investigation on finding a row with an identifier.

```
public class PersonDaoImplTest {
@Autowired
private PersonDao dao ;
@Test
public void testGetById() {
Person p1 = new Person();
```

p1.setName(“Alex”);
List<Phone> phones = new ArrayList<Phone>(); Phone ph1 = new Phone();
ph1.setNumber(“7798989138”);
Phone ph2 = new Phone();
ph2.setNumber(“7798989169”);
phones.add(ph1);
phones.add(ph2);
p1.setPhones(phones);
dao.insert(p1);

Person p2 = new Person();
p2.setName(“Fred”);
List<Phone> phones2 = new ArrayList<Phone>(); Phone phone1 = new Phone();
phone1.setNumber(“8836987”);
phones.add(phone1);
p2.setPhones(phones2);
dao.insert(p2);
Assert.assertEquals(2L,dao.getAll().size());

List<Person> persons = dao.getAll();
Person p3 = persons.get(1);

Person p4 = dao.getById(p3.getId());
Assert.assertEquals(“Alex”,p4.getName());

}

}

The like data access method uses Hibernate 4 Session.get () procedure.
public class PersonDaoImpl implements PersonDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public Person getById(Integer id) {

```
public Person getById(Integer id) {
return (Person)sessionFactory.getCurrentSession().get(Person.class,id); }

}
```

Next, set the delete test procedure. First, insert two accounts of the Person and phone numbers followed by a delete of one of the records. Finally check whether the remaining one record exists in the database.

```
public class PersonDaoImplTest {
@Autowired
private PersonDao dao ;
@Test
public void testDelete() {
Person p1 = new Person();
p1.setName("Alex");
List<Phone> phones = new ArrayList<Phone>(); Phone ph1 = new Phone();
ph1.setNumber("7798989138");
Phone ph2 = new Phone();
ph2.setNumber("7798989169");
phones.add(ph1);
phones.add(ph2);
p1.setPhones(phones);
dao.insert(p1);


Person p2 = new Person();
p2.setName("Fred");
List<Phone> phones2 = new ArrayList<Phone>(); Phone phone1 = new
Phone();
phone1.setNumber("8836987");
phones.add(phone1);
p2.setPhones(phones2);
dao.insert(p2);
Assert.assertEquals(2L,dao.getAll().size());


List<Person> persons = dao.getAll();
Person p = persons.get(1);
dao.delete(p);
```

```
Assert.assertEquals(1L,dao.getAll().size());

                                                    }

                                                    }
```

The delete task uses the Hibernate 4 Session.delete () method as shown in the below code.

```
public class PersonDaoImpl implements PersonDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void delete(Person person) {
sessionFactory.getCurrentSession().delete(person); }

                                                    }
```

We will look at the update test method which inserts a record, updates the row, and tests whether the count is the same and the modernize operational.

```
public class PersonDaoImplTest {
@Autowired
private PersonDao dao ;
@Test
public void testUpdate() {
Person p1 = new Person();
p1.setName(“Alex”);
List<Phone> phones = new ArrayList<Phone>(); Phone ph1 = new Phone();
ph1.setNumber(“7798989138”);
Phone ph2 = new Phone();
ph2.setNumber(“7798989169”);
phones.add(ph1);
phones.add(ph2);
p1.setPhones(phones);
dao.insert(p1);


Assert.assertEquals(1L,dao.getAll().size());


List<Person> persons = dao.getAll();
```

```
List<Person> persons = dao.getAll();
Person p = persons.get(0);
p.setName(“Jhon”);
List<Phone> phones1 = new ArrayList<Phone>(); Phone ph3 = new Phone();
ph1.setNumber(“111111111”);
Phone ph4 = new Phone();
ph2.setNumber(“222222222”);
Phone ph5 = new Phone();
ph2.setNumber(“333333333”);
phones1.add(ph3);
phones1.add(ph4);
phones1.add(ph5);
p.setPhones(phones1);
dao.update(p);
persons = dao.getAll();
Person p2=persons.get(0);
Assert.assertEquals(“Jhon”,p2.getName());
Assert.assertEquals(3L, p2.getPhones().size()); }

}
```

The update data access action is offered below which practices Hibernate 4 Session.merge () operation.

```
public class PersonDaoImpl implements PersonDao{
@Autowired
private SessionFactory sessionFactory ;
@Override
public void update(Person person) {
sessionFactory.getCurrentSession().merge(person); }

}
```

Develop the business service tier

We will now jump to the business service tier. First, examine the test for fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class PersonServiceImplTest {
```

```
public class PersonServiceImplTest {
@Autowired
private PersonService service;
@Test
public void testFindAll(){
Assert.assertEquals(0L, service.findAll().size()); }

                                                    }
```

Then, look at how to program the equivalent method in the business service.

```
@Service
@Transactional
public class PersonServiceImpl implements PersonService{
@Autowired
private PersonDao dao;
@Autowired
private PersonMapper mapper;
@Override
public List<PersonDto> findAll() {
List<Person> persons = dao.getAll();
List<PersonDto> personDtos = new ArrayList<PersonDto>(); if(null !=persons)
{
for(Person person : persons)
{personDtos.add(mapper.mapEntityToDto(person));}

                                                    }

return personDtos;

                                                    }

                                                    }
```

As shown previously, we have a mapper for converting data access objects into entities and vice-versa, which has two operations displayed in the following code. Note that the Phone identifier is generated every time it is rearranged.

```
@Component
public class PersonMapper {
public Person mapDtoToEntity(PersonDto personDto){
Person person = new Person();
if(null !=personDto.getId()){person.setId(personDto.getId());}
```

```
if(null !=personDto.getId()){person.setId(personDto.getId());}
if(null !=personDto.getName()){person.setName(personDto.getName());}
if(null !=personDto.getNumbers()&& personDto.getNumbers().size() > 0){
List<Phone> phoneList = new ArrayList<Phone>();
if(personDto.getNumbers()!=null && personDto.getNumbers().size()>0){
for(String number : personDto.getNumbers()){
Phone phone = new Phone();
phone.setNumber(number);
phoneList.add(phone);}

                                                    }

person.setPhones(phoneList);

                                                    }

return person;

                                                    }

public PersonDto mapEntityToDto(Person person){
PersonDto personDto = new PersonDto();
if(null !=person.getId()){personDto.setId(person.getId());}
if(null !=person.getName()){personDto.setName(person.getName());}
if(null !=person.getPhones() && person.getPhones().size() >0){
List<String> phones = new ArrayList<String>(); for(Phone phone :
person.getPhones()){phones.add(phone.getNumber());}
personDto.setNumbers(phones);

                                                    }

return personDto;

                                                    }

                                                    }
```

Proceeding to the create test operation which creates an instance of Person object having two phone numbers. We retrieve the records with find all records operation and check whether one instance of Person and two instances of phone numbers exists.

```
public class PersonServiceImplTest {
@Autowired
private PersonService service;
@Test
public void testCreate(){
PersonDto personDto1 = new PersonDto();
personDto1.setName(“Alex”);
List<String> phones = new ArrayList<String>(); phones.add(“9158798405”);
phones.add(“7798989134”);
personDto1.setNumbers(phones);
service.create(personDto1);
Assert.assertEquals(1L, service.findAll().size());
List<PersonDto> personDtos = service.findAll(); PersonDto dto =
personDtos.get(0);
Assert.assertEquals(2L, dto.getNumbers().size()); }

                                                    }
```

The compliant method in the business service tier collects a data access object, transforms the corresponding entity and carries it to the data access tier for storing in the database.

```
public class PersonServiceImpl implements PersonService{
@Autowired
private PersonDao dao;
@Autowired
private PersonMapper mapper;
@Override
public void create(PersonDto personDto) {
dao.insert(mapper.mapDtoToEntity(personDto));

                                                    }

                                                    }
```

The following test operation locates a row with a given identifier.

```
public class PersonServiceImplTest {
@Autowired
private PersonService service;
@Test
public void testFindById() {
```

```
PersonDto personDto1 = new PersonDto();
personDto1.setName("Alex");
List<String> phones = new ArrayList<String>(); phones.add("9158798405");
phones.add("7798989134");
personDto1.setNumbers(phones);
service.create(personDto1);


List<PersonDto> personDtos = service.findAll(); PersonDto dto =
personDtos.get(0);
PersonDto dto1 = service.findById(dto.getId()); Assert.assertEquals("Alex",
dto1.getName());
Assert.assertEquals(2L, dto1.getNumbers().size()); }

                                                    }
```

The business service operation to find a row with a given identifier would be established now. The identifier is passed to the data access operation related to find by identifier method. When a matching row is retrieved, the entity is converted to a data transfer object using the mapper and returned.

```
public class PersonServiceImpl implements PersonService{
@Autowired
private PersonDao dao;
@Autowired
private PersonMapper mapper;
@Override
public PersonDto findById(Integer id) {
Person person = dao.getById(id);
if(null !=person){return mapper.mapEntityToDto(person);}
return null;

                                                    }

                                                    }
```

The exclude test operation shown in the following code receives a particular identifier and removes the approving row from the database. The test method first creates two specimens, removes one example and checks whether there is the sole result in the record.

```
public class PersonServiceImplTest {
@Autowired
private PersonService service;
@Test
public void testRemove() {
PersonDto personDto1 = new PersonDto();
personDto1.setName(“Alex”);
List<String> phones = new ArrayList<String>(); phones.add(“9158798405”);
phones.add(“7798989134”);
personDto1.setNumbers(phones);
service.create(personDto1);


PersonDto personDto2 = new PersonDto();
personDto2.setName(“Fred”);
List<String> phones2 = new ArrayList<String>(); phones2.add(“9158798408”);
phones2.add(“7798989139”);
personDto2.setNumbers(phones2);
service.create(personDto1);
Assert.assertEquals(2L, service.findAll().size());
List<PersonDto> personDtos = service.findAll(); PersonDto deletePerson =
personDtos.get(1);
service.remove(deletePerson.getId());
Assert.assertEquals(1L, service.findAll().size()); }

                                                    }
```

The affiliated business service method takes an identifier and removes the distinctive instance from the index over the data access tier procedure.

```
public class PersonServiceImpl implements PersonService {
@Autowired
private PersonDao dao;
@Autowired
private PersonMapper mapper;
@Override
public void remove(Integer id) {
Person person = dao.getById(id);
if(null !=person){dao.delete(person);}

                                                    }
```

}

The terminating test operation is edit operation which creates an occurrence, edits the phone numbers and then checks if the edit was in force.

```
public class PersonServiceImplTest {
@Autowired
private PersonService service;
@Test
public void testEdit(){
PersonDto personDto1 = new PersonDto();
personDto1.setName(“Alex”);
List<String> phones = new ArrayList<String>(); phones.add(“9158798405”);
phones.add(“7798989134”);
personDto1.setNumbers(phones);
service.create(personDto1);
Assert.assertEquals(1L, service.findAll().size());
List<PersonDto> personDtos = service.findAll(); PersonDto dto =
personDtos.get(0);
phones = new ArrayList<String>();
phones.add(“9158798406”);
phones.add(“9158798407”);
phones.add(“9158798408”);
dto.setNumbers(phones);
service.edit(dto);
Assert.assertEquals(3L, service.findAll().get(0).getNumbers().size()); }
```

}

The edit business service operation substitutes the call to the data access update method after makeover with the mapper method. Note that the edit operation does not have the identifier of the Phone entity and hence generates a new one for every call.

```
public class PersonServiceImpl implements PersonService{
@Autowired
private PersonDao dao;
@Autowired
private PersonMapper mapper;
@Override
public void edit(PersonDto personDto) {
```

```
dao.update(mapper.mapDtoToEntity(personDto));

                                                    }

                                                    }
```

Develop the presentation tier

The concluding tier in the batch is the REST-based Spring Controller for which we will write the test first. The test covers a create, then edit, followed by a remove action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class PersonControllerTest {
private Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
hh:mm:ss”).create(); @Resource
private WebApplicationContext webApplicationContext; private MockMvc
mockMvc;
@Before
public void setUp() {
mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build(); }
@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                                    }

public void testCreate() throws Exception {
PersonDto personDto1 = new PersonDto();
personDto1.setName(“Alex”);
List<String> phones = new ArrayList<String>(); phones.add(“9158798405”);
phones.add(“7798989134”);
personDto1.setNumbers(phones);
String json = gson.toJson(personDto1);
```

```
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post("onetomanyunidirectionalcreate");
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}
public void testUpdate() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("onetomanyunidirectionalfindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<PersonDto>>() {}.getType(); List<PersonDto> personDtoList
= gson.fromJson(response2, listType); PersonDto personDto =
personDtoList.get(0);
personDto.setName("Jhon Flex");
String json2 = gson.toJson(personDto);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("onetomanyunidirectionaledit");
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}
public void testDelete() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("onetomanyunidirectionalfindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<PersonDto>>() {}.getType(); List<PersonDto> personDtoList
= gson.fromJson(response2, listType); PersonDto personDto2 =
personDtoList.get(0);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("onetomanyunidirectionalremove/"+personDto2
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}
```

}

The consequential REST, Spring Controller, will be linking with the business

service tier. The business service tier will link to the database via the data access tier. The platform is similar to the mock service, with the change that the code is working together with the real database and not the in memory journal.

```
@Controller
@RequestMapping(value=”/onetomanyunidirectional”) public class
PersonController {
@Autowired
private PersonService service;
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<PersonDto> findAll(){
return service.findAll();

                                                    }

@RequestMapping(value=”/findById/{personid}”,
method=RequestMethod.GET) public @ResponseBody PersonDto
findById(@PathVariable(“personid”) Integer personid){
return service.findById(personid);

                                                    }

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void creat(@RequestBody PersonDto personDto){
service.create(personDto);

                                                    }

@RequestMapping(value=”/remove/{personid}”,
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“personid”) Integer personid){
service.remove(personid);

                                                    }

@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody PersonDto personDto){
service.edit(personDto);
```

}

}

The final step is the incorporation of the REST service with the real user interface. Please note that the request mapping changes from "*onetomanyunidirectional*mock" to "/onetomanyunidirectional". The final task encompasses writing the real user interface using the mock user interface as a reference.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a one-to-many unidirectional association
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 7. One-to-Many Bidirectional Relationship A one-to-many bidirectional relationship is a one-to-many relationship which has reference from parent to child and also vice versa. We will survey the accustomed flow of starting with the user interface shadowed by the service development and assimilating both of the components at the close of the chapter.

Domain Model

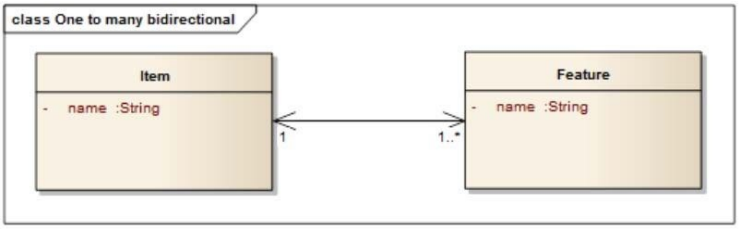

Figure 7-1: A one-to-many bidirectional relationship In this chapter, we will demeanor a one-to-many bidirectional relationship, as shown in Figure 7-1. The Item entity has an identifier and a name field of type String. The Feature entity also has an identifier and a number String field. In an instance of Item, there must be one or more instances of Feature associated. Similarly, in an instance of Feature there must be one instance of Item linked. Since the relationship is bidirectional you can traverse the list of Features from Item and also an Item from an instance of Feature. We will survey the classic channel of writing with the mock user interface firstly and then the unpretentious service.

Develop a User Interface

The three significant tasks we scrutinize during the evolution of the mock user interface are the creation of the data transfer object, the mock service and the mock user interface.

Develop the data transfer object

The Item data transfer object has an identifier, name of the Item, and a list of String Feature names. The identifier of the Feature object is hidden from the caller user interface since we wanted to make the API call much simpler for the user interface team. Also, the idea is not to expose the database design because of the complexity and not to get involved in managing the identifiers of the child object.

```
public class ItemDto {
private Integer id;
private String name;
private List<String> featureList;

// getters and setters

}
```

Develop the mock service

The mock service will be interacting with an in-memory customary database for saving the data, which will be consumed by the mock user interface. We will originate with add functionality. We have a list of ItemDto, and with every new topic, one instance of ItemDto having an item name and feature names is added to the list. A Singleton design pattern using enum is charted for the in-memory store implementation. The identifier field inducts with 1 and progresses by 1 for every new count.

```
public enum ItemInMemoryDB {
INSTANCE;
private static List<ItemDto> list = new ArrayList<ItemDto>(); private static
Integer lastId = 0;
public Integer getId() {
return ++lastId;

}
```

```
public void add(ItemDto itemDto){
itemDto.setId(getId());
list.add(itemDto);

                                                    }

                                                    }
```

Now let’s watch the equivalent mock service implementation. The create method collects an instance of ItemDto from the user interface and enhances the list via the mock file.

```
@Controller
@RequestMapping(value=”onetomanybidirectionalmock”) public class
ItemMockController {
@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void creat(@RequestBody ItemDto itemDto){
ItemInMemoryDB.INSTANCE.add(itemDto);

                                                    }

                                                    }
```

Next we will ensue with the find all rows and find with identifier fragment.

```
public enum ItemInMemoryDB {
public List<ItemDto> findAll(){
return list;

                                                    }

public ItemDto findById(Integer id) {
for(ItemDto itemDto : list){
if(itemDto.getId() == id)
return itemDto ;

                                                    }

return null;

                                                    }
```

}

In the mock package, then find all rows harvests a list of ItemDto and find with identifier conventions an instance of ItemDto.

```
public class ItemMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<ItemDto> findAll(){
return ItemInMemoryDB.INSTANCE.findAll();

}

@RequestMapping(value=”/findById/{itemid}”, method=RequestMethod.GET)
public @ResponseBody ItemDto findById(@PathVariable(“itemid”) Integer
itemid){
return ItemInMemoryDB.INSTANCE.findById(itemid);

}

}
```

Next, we will instrument the remove operation which impounds a specific row with a recorded identifier.

```
public enum ItemInMemoryDB {
public void remove(Integer id) {
ItemDto dto = null;
for(ItemDto itemDto : list){
if(itemDto.getId() == id){
dto = itemDto;
break;

}

}

if(null != dto){list.remove(dto);}

}

}
```

In the equivalent mock service, the omit operation removes a specific row with the distinct identifier supplied as a response to the url.

```
public class ItemMockController {
@RequestMapping(value=”/remove/{itemid}”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“itemid”) Integer itemid){
ItemInMemoryDB.INSTANCE.remove(itemid);

}

}
```

Most latterly we yield the edit operation responsible for renovating a row.

```
public enum ItemInMemoryDB {
public void edit(ItemDto itemDto){
for(ItemDto dto : list){
if(dto.getId() == itemDto.getId()){
dto.setName(itemDto.getName());
dto.setFeatureList(itemDto.getFeatureList());

}

}

}

}
```

The exclusive edit operation accepts an altered copy of the ItemDto and refurbishes the specific row with the deviances.

```
public class ItemMockController {
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody ItemDto itemDto){
ItemInMemoryDB.INSTANCE.edit(itemDto);

}

}
```

Develop the mock user interface

We inspected in the previous section how to foster the mock service with an in-memory store. Now we decide how to write the user interface which will swallow this mock service. The user interface will have the means to add, edit, delete and show all records related to the Item, including the Feature names. A screenshot of the user interface is shown in Figure 7-2.

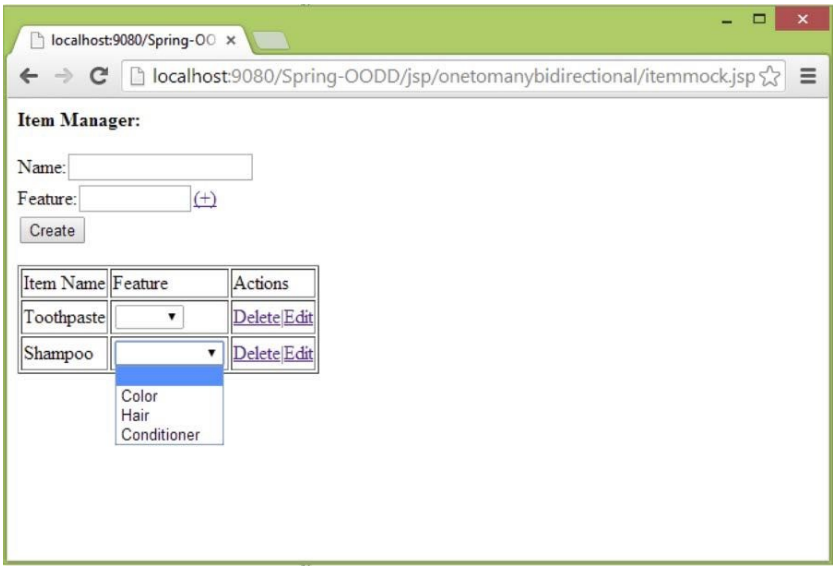

Figure 7-2: A screenshot of the Item and Feature user interface The JSP body is the opening page which becomes assessed and after that the Javascript actions gets called. We will outline the torrent with the Javascript requests.

```
<body onload=”loadObjects()”>
<div id=”container”>
<div>
<p>
<b>Item Manager:</b>
</p>
</div>
<form method=”post” id=”itemForm”>
<div id=”itemDiv”>
<div id=”nameGroup”>
<label>Name:<input type=”text” name=”name” id=”name”></label> <label>
<input type=”hidden” name=”itemid” id=”itemid”></label> </div>
</div>
```

<label>Feature:<input type=”text” name=”feature” id=”feature” size=”10” maxlength=”12”>(+) </label> <label id=”removeLink_1”></label>

<input type=submit value=”Create” id=”subButton” onclick=”return methodCall()”> </form>

</body>

To begin with, we will learn the JavaScript loadObjects method which is called when the page is ready with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDonetomanybidirectional/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

                                                {

processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown)

                                                {

document.getElementById(“name”).value=””;
alert(“Error Status Load Objects:”+textStatus);

                                                }

                                               });

return false;

                                                }
```

The processResponseData JavaScript method is called when the above service call is active. Also the generateTableData Javascript method is shown which creates the table structure.

```
function processResponseData(objects){
var dyanamicTableRow="<table border=1>"+"<tr>" +"<td>Item
Name</td>"+"<td>Feature</td>"+"<td>Actions</td>"+"</tr>"; var
dataRow="";
$.each(objects, function(itemno, object)
{dataRow=dataRow+generateTableData(object);});
dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("itemFormResponse").innerHTML=dyanamicTableR
}
function generateTableData(object){
var features = object.featureList;
var dataRow="<tr>" +"<td>" +object.name+"</td>";dataRow =
dataRow+"<td>"+"<select>"+"<option>"+"</option>"; if(features!=null &&
features.length > 0){for(var index=0;index<features.length;index++) { var
feature = features[index];dataRow = dataRow+"<option
value="+feature+">"+feature+"</option>";}}
dataRow=dataRow+"</select></td>";
dataRow=dataRow+"<td>" +"<a href=#
onclick=deleteObject("+object.id+")>Delete</a>"+
"|<a href=# onclick=editObject("+object.id+")>Edit</a>"+"</td>"+"</tr>";
return dataRow;

                                    }
```

We have seen the page loading technique. If you diagnose how the JSP fragment works, you will notice Javascript method methodCall is being called which is appropriate for both create and update functionality.

```
function methodCall(){
var buttonValue = document.getElementById("subButton").value;
if(buttonValue=="Create"){
create();
}else if(buttonValue=="Update"){
update();

                                    }

return false;
```

return false;

}

We will recognize the create Javascript method which gathers the values from the text boxes item name and feature names and passes the concept as an ItemDto without an identifier.

```
function create(){
var name = $("#name").val();
var numbers = null;
var flag= false;
var feature=document.getElementsByName("feature"); if(feature.length > 0)
{numbers =  new Array();for(var i=0;i<feature.length;i++) {
if(trim(feature[i].value)!=""){
numbers.push(feature[i].value);flag=true;

}

}

}

if(flag==false){numbers=null;}
var formData={"name":name,"featureList":numbers};
$.ajax({url : "Spring-OODDonetomanybidirectional/mock/create",type:
"POST", data : JSON.stringify(formData),beforeSend: function(xhr)
{xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json");}, success:
function(data, textStatus, jqXHR){
document.getElementById("name").value="";
document.getElementById("subButton").value="Create"; loadObjects();
initiateItemBoxGroup();

},
error: function (jqXHR, textStatus, errorThrown)

{

document.getElementById("name").value="";
initiateItemBoxGroup();
alert("Error Status Create:"+textStatus);
```

`  `

}

});

return false;

}

We will also look at the initiateItemBoxGroup Javascript method which rearranges the feature names user interface area.

```
function initiateItemBoxGroup(){
var itemBoxGroupDiv = document.getElementById(“itemBoxGroup”);
itemBoxGroupDiv.innerHTML=””;
var itemeDiv = document.createElement(‘div’);
itemeDiv.setAttribute(“id”,”itemBox_1”);
var itemBox_1 = “<label>Feature:<input type=’text’ name=’feature’
id=’feature’ size=’10’ maxlength=’12’><a href=#
onclick=javascript:addMoreFeature()>(+)</a></label>”; var
removeLink_1=”<label id=’removeLink_1’></label>”; var setUrl =
itemBox_1+removeLink_1;
itemeDiv.innerHTML=setUrl;
itemBoxGroupDiv.appendChild(itemeDiv);
```

}

Next we will look at the updated Javascript method which keeps an ItemDto including an identifier, item name and feature names.

```
function update(){
var name = $(“#name”).val();
var id = +$(“#itemid”).val();
var flag= false;
var feature=document.getElementsByName(“feature”); if(feature.length > 0)
{numbers =  new Array();for(var i=0;i<feature.length;i++)
{if(trim(feature[i].value)!=””){numbers.push(feature[i].value);flag=true;}}}
if(flag==false){numbers=null;}
var formData={“id”:id,”name”:name,”featureList”:numbers}; $.ajax({
url : “Spring-OODDonetomanybidirectional/mock/edit”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
```

xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

document.getElementById(“name”).value=””;
initiateItemBoxGroup();
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown)

{

document.getElementById(“name”).value=””;
initiateItemBoxGroup();
alert(“Error Status Update:”+textStatus);

}

});

return false;

}

The editObject and viewObject Javascript method calls are used to retrieve the selected ItemDto object and display for editing, including the feature names.
function editObject(itemid){
editurl=”*Spring-OODD*onetomanybidirectional/mock/findById/”+itemid; var itemForm={id:itemid};
$.ajax({url : editurl,type: “GET”,data : itemForm,dataType: “json”, success: function(data, textStatus, jqXHR){
viewObject(data);
document.getElementById(“subButton”).value=”Update”;}, error: function (jqXHR, textStatus, errorThrown){alert(“Error Status Find Object:”+textStatus);}

});

}

}

function viewObject(data){
var index = 1;
document.getElementById(“name”).value=data.name;
document.getElementById(“itemid”).value=data.id;
var numbers = data.featureList;
if(numbers!=null){
initiateItemBoxGroup();
var itemBoxGroupDiv = document.getElementById(“itemBoxGroup”); for(var cnt=0;cnt<numbers.length;cnt++){
if(cnt==0){
var featurelist= document.getElementsByName(“feature”);
featurelist[cnt].value=numbers[cnt];
}else{
index++;
var itemDiv = document.createElement(‘div’);
itemDiv.setAttribute(“id”,”itemBox_”+index);
var remId=”removeLink_”+index;
var removeLinkUrl = “|<label id=”+remId+”>(-)</label>”; itemDiv.innerHTML = “<label>Feature:”+”<input type=’text’ id=’feature’ name=’feature’ maxlength=’12’ size=’10’ value=’”+numbers[cnt]+”’/>
</label>”+removeLinkUrl; itemBoxGroupDiv.appendChild(itemDiv);

}

}

}

}

To complete, we will look at the delete operation where the item identifier is passed as a response and the related Item along with Feature information is removed.
function deleteObject(itemid){
var itemForm={id:itemid};
delurl=”*Spring-OODD*onetomanybidirectional/mock/remove/”+itemid; $.ajax({
url : delurl,
type : “POST”

```
type: "POST",
data : itemForm,
dataType: "json",
success: function(data, textStatus, jqXHR){
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown){
alert("Error Status Delete:"+textStatus);

}

});

}
```

Also, as a last footnote, the feature area addition and deletion of text boxes so that multiple features can be managed is described. This is done with the addMoreFeature and removeFeature Javascript method calls.

```
var counter = 1;
function addMoreFeature(){
counter++;
var itemeDiv = document.createElement('div');
itemeDiv.setAttribute("id","itemBox"+"_"+counter); var
remId="removeLink_"+counter;
var removeLinkUrl = "|<label id="+remId+"><a href='#'
onclick='removeFeature("+counter+")'>(-)</a></label>"; var
itemBoxGroupDiv = document.getElementById("itemBoxGroup");
itemeDiv.innerHTML = "<label>Feature:"+"<input type='text' id='feature'
name='feature' maxlength='12' size='10'/>" +
"</label>"+removeLinkUrl;
itemBoxGroupDiv.appendChild(itemeDiv);

}

function removeFeature(index){
var featurelist=document.getElementsByName("feature"); var size =
featurelist.length;
if(size==1){counter=1;return false;}
var itemBoxGroupDiv = document.getElementById("itemBoxGroup");
itemBoxGroupDiv.removeChild(document.getElementById("itemBox"+"_"+ind
```

}

Develop the Service

The following step is the actual service development which involves four main steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST capacity. We will begin with the entity and go bottom up to the REST service.

Develop the resource (entity) tier As an implementation technique, we are ensuing the test driven development approach, hence we will charge the test for the entity to begin with and then the entity actual development. First, we create two instances of Item with Feature list and then eventually check for two instances of Item. Later, we do an update and test whether the changes echoed. Lastly, we detach one occurrence and test whether the count is lonely.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ItemTest {
@Autowired
private SessionFactory sessionFactory;
@Test
public void testCRUD(){
Item item1 = new Item();
item1.setName(“Book”);
List<Feature> list = new ArrayList<Feature>(); Feature feature1 = new
Feature();
feature1.setName(“Core Java”);
feature1.setItem(item1);
Feature feature2 = new Feature();
feature2.setName(“Spring”);
```

```
feature2.setName("Spring");
feature2.setItem(item1);
list.add(feature1);
list.add(feature2);
item1.setFeatures(list);


sessionFactory.getCurrentSession().save(item1);


Item item2 = new Item();
item2.setName("Bags");
List<Feature> list2 = new ArrayList<Feature>(); Feature feature3 = new
Feature();
feature3.setName("SkyBags");
feature3.setItem(item2);
Feature feature4 = new Feature();
feature4.setName("WildCraft");
feature4.setItem(item2);
list2.add(feature3);
list2.add(feature4);
item2.setFeatures(list2);


sessionFactory.getCurrentSession().save(item2);


@SuppressWarnings("unchecked")
List<Item> items = sessionFactory.getCurrentSession() .createQuery("select
item from Item item order by item.id desc").list();
Assert.assertEquals(2L,items.size());


Item item = items.get(0);
item.setName("BookList");
sessionFactory.getCurrentSession().update(item);
```

```
@SuppressWarnings(“unchecked”)
List<Item> itemslist = sessionFactory.getCurrentSession() .createQuery(“select item from Item item order by item.id desc”).list(); Item pItem = itemslist.get(0);
Assert.assertEquals(“BookList”,pItem.getName());


sessionFactory.getCurrentSession().delete(pItem);
@SuppressWarnings(“unchecked”)
List<Item> ppItem =  sessionFactory.getCurrentSession() .createQuery(“select item from Item item order by item.id desc”).list();
Assert.assertEquals(1L,ppItem.size());

                                                    }

                                                    }
```

We will explore the Feature entity which contains an identifier and a name String field. Since the entity is on the inverse side and is a bidirectional association, a reference to Item is mentioned as a @ManyToOne association.

```
@Entity
@Table(name=”FEATURE”)
public class Feature {
private Integer id;
private String name;
private Item item;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”,length=20)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }

@Column(name=”name”,length=50)
public String getName() {
```

```
public String getName() {
return name;

                                                    }

public void setName(String name) {
this.name = name;

                                                    }

@ManyToOne
   @JoinColumn(name="ITEM_ID", nullable=false) public Item getItem() {
return item;

                                                    }

public void setItem(Item item) {
this.item = item;

                                                    }

                                                    }
```

We are developing a one-to-many bidirectional relationship. An instance of Item must have one or more instances of Feature. Since the relationship is bidirectional, reference to a list of Feature is stated. We also practice orphanRemoval=true because if a set of Features is reassigned, the old Features will be deleted. The Item is the owner and a reference to the Feature is maintained using @OneToMany annotation.

```
@Entity
@Table(name="ITEM")
public class Item {
private Integer id;
private String name;
private List<Feature> features;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="ID",length=20)
public Integer getId() {
return id;
```

```
                                        }

public void setId(Integer id) {
this.id = id;

                                        }

@Column(name="name",length=50)
public String getName() {
return name;

                                        }

public void setName(String name) {
this.name = name;

                                        }

@OneToMany(mappedBy="item",cascade=CascadeType.ALL,orphanRemoval=
public List<Feature> getFeatures() {
return features;

                                        }

public void setFeatures(List<Feature> features) {
this.features = features;

                                        }

}
```

Table structure

```
DROP TABLE `feature`;
DROP TABLE `item`;
CREATE TABLE `item` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `name` varchar(50)
DEFAULT NULL,
  PRIMARY KEY (`ID`)

                                        `
```

);

```
CREATE TABLE `feature` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `name` varchar(50)
DEFAULT NULL,
  `ITEM_ID` int(11) NOT NULL,
  PRIMARY KEY (`ID`),
  KEY (`ITEM_ID`),
  CONSTRAINT FOREIGN KEY (`ITEM_ID`) REFERENCES `item` (`ID`) );
```

Develop the data access tier

Next we will watch the data access tier. We will start with the test case for the get all records method.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ItemDaoImplTest {
@Autowired
private ItemDao dao;
@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size());

}

}
```

We will grasp how to write the emulating data access operation for the test directly above. The Hibernate Session.createQuery method is used with an SQL to retrieve all the records from the Item entity.

```
@Repository
@Transactional
public class ItemDaoImpl implements ItemDao {
@Autowired
private SessionFactory sessionFactory ;
@SuppressWarnings(“unchecked”)
@Override
public List<Item> getAll() {
return (List<Item>)sessionFactory.getCurrentSession()
```

```
return (List<Item>)sessionFactory.getCurrentSession().
createQuery("select item from Item item order by item.id desc ").list(); }

}
```

Consequently, let's now observe the test for insert operation. We insert a single row into the Item entity along with Features and check if the results match a solitary record.

```
public class ItemDaoImplTest {
@Autowired
private ItemDao dao;
@Test
public void testInsert(){
Item item = new Item();
item.setName("Book");
List<Feature> list = new ArrayList<Feature>(); Feature feature1 = new
Feature();
feature1.setName("Core Java");
feature1.setItem(item);
Feature feature2 = new Feature();
feature2.setName("Spring");
feature2.setItem(item);
list.add(feature1);
list.add(feature2);
item.setFeatures(list);
dao.insert(item);
Assert.assertEquals(1L, dao.getAll().size());

}

}
```

The matching insert operation just requests the Hibernate Session.save method. The Item entity identifier gets set dependably along with the identifier of the Feature.

```
public class ItemDaoImpl implements ItemDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public void insert(Item item) {
```

```
sessionFactory.getCurrentSession().save(item);

                                                    }

}
```

We will then look at the test of finding a row with an identifier.

```
public class ItemDaoImplTest {
@Autowired
private ItemDao dao;
@Test
public void testGetById() {
Item item1 = new Item();
item1.setName(“Book”);
List<Feature> list = new ArrayList<Feature>(); Feature feature1 = new
Feature();
feature1.setName(“Core Java”);
feature1.setItem(item1);
Feature feature2 = new Feature();
feature2.setName(“Spring”);
feature2.setItem(item1);
list.add(feature1);
list.add(feature2);
item1.setFeatures(list);
dao.insert(item1);


Assert.assertEquals(1L, dao.getAll().size());


List<Item> items = dao.getAll();
Item item2 = items.get(0);
Item item3 = dao.getById(item2.getId());


Assert.assertEquals(“Book”, item3.getName());
Assert.assertEquals(2L, item3.getFeatures().size()); }

                                                    }
```

The corresponding data access method which uses Hibernate 4 Session.get () procedure is unpretentious.

```
public class ItemDaoImpl implements ItemDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public Item getById(Integer id) {
return (Item) sessionFactory.getCurrentSession().get(Item.class, id); }

}
```

We will set the delete test procedure at this stage. First, we insert two accounts of the Item and features followed by a delete of one of the records and then test whether an isolated record survives in the stockpile.

```
public class ItemDaoImplTest {
@Autowired
private ItemDao dao;
@Test
public void testDelete() {
Item item1 = new Item();
item1.setName(“Book”);
List<Feature> list = new ArrayList<Feature>(); Feature feature1 = new
Feature();
feature1.setName(“Core Java”);
feature1.setItem(item1);
Feature feature2 = new Feature();
feature2.setName(“Spring”);
feature2.setItem(item1);
list.add(feature1);
list.add(feature2);
item1.setFeatures(list);
dao.insert(item1);


Item item2 = new Item();
item2.setName(“Bags”);
List<Feature> list2 = new ArrayList<Feature>(); Feature feature3 = new
Feature();
feature3.setName(“SkyBags”);
```

```
feature3.setItem(item2);
Feature feature4 = new Feature();
feature4.setName(“WildCraft”);
feature4.setItem(item2);
list2.add(feature3);
list2.add(feature4);
item2.setFeatures(list2);
dao.insert(item2);
Assert.assertEquals(2L, dao.getAll().size());


List<Item> items = dao.getAll();
Item item = items.get(0);
dao.delete(item);
Assert.assertEquals(1L, dao.getAll().size());

                                                    }

                                                    }
```

The delete task uses the Hibernate 4 Session.delete () method.

```
public class ItemDaoImpl implements ItemDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public void delete(Item item) {
sessionFactory.getCurrentSession().delete(item);

                                                    }

                                                    }
```

Lastly, we will use the update test method which inserts two records, updates the row and tests whether the count is the same and the renovation was working.

```
public class ItemDaoImplTest {
@Autowired
private ItemDao dao;
@Test
public void testUpdate() {
Item item1 = new Item();
```

```
Item item1 = new Item();
item1.setName(“Book”);
List<Feature> list = new ArrayList<Feature>(); Feature feature1 = new
Feature();
feature1.setName(“Core Java”);
feature1.setItem(item1);
Feature feature2 = new Feature();
feature2.setName(“Spring”);
feature2.setItem(item1);
list.add(feature1);
list.add(feature2);
item1.setFeatures(list);
dao.insert(item1);


Item item2 = new Item();
item2.setName(“Bags”);
List<Feature> list2 = new ArrayList<Feature>(); Feature feature3 = new
Feature();
feature3.setName(“SkyBags”);
feature3.setItem(item2);
Feature feature4 = new Feature();
feature4.setName(“WildCraft”);
feature4.setItem(item2);
list2.add(feature3);
list2.add(feature4);
item2.setFeatures(list2);
dao.insert(item2);


List<Item> items = dao.getAll();
Item item = items.get(1);
Feature feature = item.getFeatures().get(1);
feature.setName(“Spring 3.2”);
dao.update(item);
items = dao.getAll();
item = items.get(1);
Assert.assertEquals(“Spring 3.2”,item.getFeatures().get(1).getName()); }

}
```

}

The update data access action is obtainable below, which practices Hibernate 4 Session.merge () operation.

```
public class ItemDaoImpl implements ItemDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public void update(Item item) {
sessionFactory.getCurrentSession().merge(item);

}

}
```

Develop the business service tier

We will now go to the business service tier. First, we will examine the test for fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ItemServiceImplTest {
@Autowired
private ItemService service;
@Test
public void testFindAll(){
Assert.assertEquals(0L, service.findAll().size()); }

}
```

Let us contemplate how to program the corresponding method in the business service.

```
@Service
@Transactional
public class ItemServiceImpl implements ItemService{
@Autowired
private ItemDao dao;
@Autowired
private ItemMapper mapper;
```

```
private ItemMapper mapper;
@Override
public List<ItemDto> findAll() {
List<Item> items = dao.getAll();
List<ItemDto> itemDtos = new ArrayList<ItemDto>(); if(null !=items){
for (Item item : items) {
itemDtos.add(mapper.mapEntityToDto(item));

                                                    }

                                                    }

return itemDtos;

                                                    }

                                                    }
```

As indicated earlier, we have a mapper for translating data access objects into entities and vice-versa, which usually has two notable operations displayed underneath. One thing to note here is that the Feature identifier is produced every time it is repositioned.

```
@Component
public class ItemMapper {
public Item mapDtoToEntity(ItemDto itemDto){
Item item = new Item();
if(null != itemDto.getId()){item.setId(itemDto.getId());}
if(null != itemDto.getName()){item.setName(itemDto.getName());}
if(null != itemDto.getFeatureList() && itemDto.getFeatureList().size()>0){
List<Feature> features = new ArrayList<Feature>(); for(String feature :
itemDto.getFeatureList()){
Feature featureObject = new Feature();
featureObject.setName(feature);
featureObject.setItem(item);
features.add(featureObject);

                                                    }

item.setFeatures(features);

                                                    }
```

```
return item ;
                                                    }

public ItemDto mapEntityToDto(Item item){
ItemDto itemDto = new ItemDto();
if(null != item.getId()){itemDto.setId(item.getId());}
if(null != item.getName()){itemDto.setName(item.getName());}
if(null != item.getFeatures() && item.getFeatures().size() > 0){
List<String> featuers = new ArrayList<String>(); for(Feature feature :
item.getFeatures()){
featuers.add(feature.getName());
                                                    }

itemDto.setFeatureList(featuers);
                                                    }

return itemDto;
                                                    }

                                                    }
```

Going on to the create test operation, which creates one occurrence of the Item object with its features and checks if the number of instances is singular with the fetch all records operation.

```
public class ItemServiceImplTest {
@Autowired
private ItemService service;
@Test
public void testCreate(){
ItemDto itemDto = new ItemDto();
itemDto.setName(“Book”);
List<String> list = new ArrayList<String>(); list.add(«Java»);
list.add(«JEE»);
list.add(“Spring”);
itemDto.setFeatureList(list);
service.create(itemDto);
```

service.create(itemDto);
Assert.assertEquals(1L, service.findAll().size()); }

}

The accommodating method in the business service tier collects a data access object, renovates to the corresponding entity, and messengers to the data access tier for loading to the database.

```
public class ItemServiceImpl implements ItemService{
@Autowired
private ItemDao dao;
@Autowired
private ItemMapper mapper;
@Override
public void create(ItemDto itemDto) {
dao.insert(mapper.mapDtoToEntity(itemDto));
```

}

}

The next test operation is finding a row with a given identifier.

```
public class ItemServiceImplTest {
@Autowired
private ItemService service;
@Test
public void testFindById() {
ItemDto itemDto = new ItemDto();
itemDto.setName(“Book”);
List<String> list = new ArrayList<String>(); list.add(«Java»);
list.add(«JEE»);
list.add(“Spring”);
itemDto.setFeatureList(list);
service.create(itemDto);
```

List<ItemDto> itemDtos = service.findAll(); ItemDto dto = itemDtos.get(0);
ItemDto dto1 = service.findById(dto.getId());
Assert.assertEquals(“Book”, dto1.getName());
Assert.assertEquals(3L, dto1.getFeatureList().size()); }

}

The analogous operation of business service to find a row with a given identifier would be recognized now. Yet again the related command by calling the data access tier and the club up to the mapper is registered.

```
public class ItemServiceImpl implements ItemService{
@Autowired
private ItemDao dao;
@Autowired
private ItemMapper mapper;
@Override
public ItemDto findById(Integer id) {
Item item = dao.getById(id);
if(null !=item){
return mapper.mapEntityToDto(item);
}
return null;
}
}
```

The eliminate test operation is next on the agenda, which receives a particular identifier and removes the appreciative row from the database. The test method first creates two samples, removes one example, and checks whether there is only one result in the database.

```
public class ItemServiceImplTest {
@Autowired
private ItemService service;
@Test
public void testRemove() {
ItemDto itemDto1 = new ItemDto();
itemDto1.setName(“Book”);
List<String> list1 = new ArrayList<String>(); list1.add(“Java”);
list1.add(“JEE”);
list1.add(“Spring”);
itemDto1.setFeatureList(list1);
```

```
service.create(itemDto1);


ItemDto itemDto2 = new ItemDto();
itemDto2.setName(“Bags”);
List<String> list2 = new ArrayList<String>(); list2.add(“SkyBags”);
list2.add(“WildCraft”);
list2.add(“Puma”);
itemDto2.setFeatureList(list2);
service.create(itemDto2);
Assert.assertEquals(2L, service.findAll().size());
List<ItemDto> itemDtos = service.findAll();
ItemDto itemDto = itemDtos.get(1);
service.remove(itemDto.getId());
itemDtos = service.findAll();
Assert.assertEquals(1L, service.findAll().size()); }

                                                    }
```

The allied business service method is established, which takes an identifier and removes the idiosyncratic instance from the directory with the data access tier technique.

```
public class ItemServiceImpl implements ItemService{
@Autowired
private ItemDao dao;
@Autowired
private ItemMapper mapper;
@Override
public void remove(Integer id) {
Item item = dao.getById(id);
if(null != item){
dao.delete(item);

                                                    }

                                                    }

}
```

The ending test operation is the edit, which creates an occurrence, edits the

features and then checks if the edit was effective.

```
public class ItemServiceImplTest {
@Autowired
private ItemService service;
@Test
public void testEdit(){
ItemDto itemDto = new ItemDto();
itemDto.setName(“Book”);
List<String> list = new ArrayList<String>(); list.add(«Java»);
list.add(«JEE»);
list.add(“Spring”);
itemDto.setFeatureList(list);
service.create(itemDto);
Assert.assertEquals(1L, service.findAll().size());
List<ItemDto> itemDtos = service.findAll(); ItemDto dto = itemDtos.get(0);
list = new ArrayList<String>();
list.add(“EJB 3.0”);
dto.setFeatureList(list);
service.edit(dto);
Assert.assertEquals(1L, service.findAll().get(0).getFeatureList().size()); }

                                                    }
```

The corresponding operation of the business service is the edit operation which alternates the call to the data access, the update method after remodeling with the mapper method. Please note that the edit operation does not have the identifier of the Feature entity and hence spawns a new one for every call.

```
public class ItemServiceImpl implements ItemService{
@Autowired
private ItemDao dao;
@Autowired
private ItemMapper mapper;
@Override
public void edit(ItemDto itemDto) {
dao.update(mapper.mapDtoToEntity(itemDto));

                                                    }

                                                    }
```

Develop the presentation tier

The finishing tier in the lot is the REST based Spring Controller for which we will write the test first. The test covers create, then edit, followed by a removal action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true)
@Transactional
public class ItemControllerTest {
private final Gson gson = new GsonBuilder().create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc = MockMvcBuilders.
<StandaloneMockMvcBuilder>webAppContextSetup
(webApplicationContext).build();

                                                }

@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                                }

public void testCreate() throws Exception {
ItemDto itemDto = new ItemDto();
itemDto.setName(“Book”);
List<String> featurelist = new ArrayList<String>(); featurelist.add(“Java”);
featurelist.add(“J2ee”);
itemDto.setFeatureList(featurelist);
String json = gson.toJson(itemDto);
```

```
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post("onetomanybidirectionalcreate");
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}
public void testUpdate() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("onetomanybidirectionalfindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<ItemDto>>() {}.getType(); List<ItemDto> itemDtos =
gson.fromJson(response2, listType); ItemDto itemDto = itemDtos.get(0);
List<String> featurelist = new ArrayList<String>(); featurelist.add("Spring
3.0");
featurelist.add("Hibernate 4.1");
itemDto.setFeatureList(featurelist);
String json2 = gson.toJson(itemDto);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("onetomanybidirectionaledit");
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}
public void testDelete() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("onetomanybidirectionalfindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<ItemDto>>() {}.getType(); List<ItemDto> itemDtos =
gson.fromJson(response2, listType); ItemDto itemDto = itemDtos.get(0);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("onetomanybidirectionalremove/"+itemDto.getI
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

}
```

The resulting REST, Spring Controller, will be linking with the business service tier which is finally through the data access tier to link to the database. The platform is comparable to the mock service, with the alteration that the code is working together with the real database and not the in memory bulletin.

```
@Controller
@RequestMapping(value=”/onetomanybidirectional”)
public class ItemController {
@Autowired
private ItemService service;
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<ItemDto> findAll(){
return service.findAll();

                                        }

@RequestMapping(value=”/findById/{itemid}”, method=RequestMethod.GET)
public @ResponseBody ItemDto findById(@PathVariable(“itemid”) Integer
itemid){
return service.findById(itemid);

                                        }

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ItemDto itemDto){
service.create(itemDto);

                                        }

@RequestMapping(value=”/remove/{itemid}”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“itemid”) Integer itemid){
service.remove(itemid);

                                        }

@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody ItemDto itemDto){
service.edit(itemDto);
```

}

}

The ultimate step in the gradient is the incorporation of the REST service with the genuine user interface. Please note that the request mapping changes from "*onetomanybidirectional*mock" to "/onetomanybidirectional". The remaining task includes writing the real user interface using the mock user interface as a reference.

Summary

In this chapter we discussed the following:

- Develop entity related to one-to-many bidirectional associations
- Develop data access tier
- Develop business service tier
- Develop a REST service controller
- Develop mock user interface
- Develop mock in-memory database.

Chapter 8. One-to-Many SelfReferencing Relationship A one-to-many selfreferencing relationship is a one-to-many relationship which has reference to itself. We will study the adapted tide of opening with the user interface stalked by the service development and embracing both mechanisms at the end of the chapter.

Domain Model

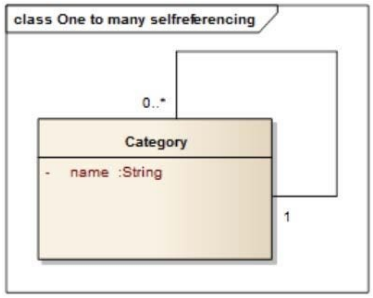

Figure 8-1: A one-to-many selfreferencing relationship In this chapter, we will conduct a one-to-many selfreferencing relationship as shown in Figure 8-1. The Category entity has an identifier and a name field of type String. In an instance of Category, there is an instance of parent Category associated. Only for the root element, there is no parent. Think of this example as a tree or a product hierarchy where Books is the root and the next level is Java, C++ and so on. Please note that the relationship is a one-to-many selfreferencing relationship, hence the parent can be shared by many children. For example, there may be more than one type of book under Books category. We will review the mock user interface first of all and then the natural service.

Develop a User Interface

The three primary tasks we look at during the development of the mock user interface are the creation of the data transfer object, the mock service and the mock user interface. We will look at these one by one.

Develop the data transfer object

The Category data transfer object has an identifier, the name of the category, and the parent category identifier. The parent can be shared by many children; hence we cannot recreate the parent every time we create a new category. Therefore, we are skipping the parent identifier, but not the parent category name. If we skip the parent category name, the parent category has to be recreated and all the related categories also have to be recreated by updating the parent category, which is a cumbersome task.

```
public class CategoryDto {
private Integer id;
private String name;
private Integer parentId;

// getters and setters

}
```

Develop the mock service

The mock service will be interacting with an in-memory custom database for persisting the data which will be used by the mock user interface. We will invent with add functionality. We have a list of CategoryDto, and with every new issue, one instance of CategoryDto having a name and parent identifier is added to the list. A Singleton design pattern using enum is registered for the in-memory collection operation. The identifier field begins with 1 and grows by 1 for every new tally.

```
public enum CategoryInMemoryDB {
INSTANCE;


private static List<CategoryDto> list = new ArrayList<CategoryDto>();
```

```
private static List<CategoryDto> list = new ArrayList<CategoryDto>();
private static Integer lastId = 0;


public Integer getId() {
return ++lastId;

                                                    }

public void add(CategoryDto categoryDto){
categoryDto.setId(getId());
list.add(categoryDto);

                                                    }

                                                    }
```

Now let's look at the corresponding mock service implementation. The create method collects an instance of CategoryDto from the user interface and increments the list via the mock service.

```
@Controller
@RequestMapping(value="/onetomanyselfreferencemock") public class
CategoryMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody CategoryDto categoryDto){
CategoryInMemoryDB.INSTANCE.add(categoryDto);

                                                    }

}
```

Next we will proceed with the find all rows and find with identifier part.

```
public enum CategoryInMemoryDB {
public List<CategoryDto> findAll(){ return list; }
public CategoryDto findById(Integer id) {
for(CategoryDto categoryDto : list){
if(categoryDto.getId() == id)
return categoryDto;

                                                    }
```

return null ;

}

}

In the mock service, find all rows produces a list of CategoryDto and find with identifier gets an instance of CategoryDto.

```
public class CategoryMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<CategoryDto> findAll(){
return CategoryInMemoryDB.INSTANCE.findAll();
```

}

```
@RequestMapping(value=”/findById/{Id}”, method=RequestMethod.GET)
public @ResponseBody CategoryDto findById(@PathVariable(“Id”) Integer Id)
{
return CategoryInMemoryDB.INSTANCE.findById(Id); }
```

}

Next, we will devise the remove operation which removes a specific row with a verified identifier.

```
public enum CategoryInMemoryDB {
public void remove(Integer id) {
CategoryDto dto = null;
for(CategoryDto categoryDto : list){
if(categoryDto.getId() ==  id) {
boolean flag = false;
for(CategoryDto categoryDto2 : list){
if (id==categoryDto2.getParentId())
flag = true;
```

}

```
if (!flag) {
dto = categoryDto;
break;
```

}

}

}

}

if(null != dto) list.remove(dto);

}

}

In the comparable mock service, the remove operation eliminates a specific row with the discrete identifier provided as a response to the url.

```
public class CategoryMockController {
@RequestMapping(value=”/remove/{Id}”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“Id”) Integer Id){
CategoryInMemoryDB.INSTANCE.remove(Id);
```

}

}

Finally we look at the edit operation accountable for modernizing a row.

```
public enum CategoryInMemoryDB {
public void edit(CategoryDto categoryDto){
for(CategoryDto dto : list){
if(categoryDto.getId() == dto.getId()){
dto.setName(categoryDto.getName());
dto.setParentId(categoryDto.getParentId());
break;
```

}

}

}

}

The select edit operation receives a transformed copy of the CategoryDto and updates the specific row with the deviations.
public class CategoryMockController {
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody CategoryDto categoryDto){
CategoryInMemoryDB.INSTANCE.edit(categoryDto); }

}

Develop the mock user interface

We tested the previous section, how to cultivate the mock service with an in-memory build. Now we resolve how to write the user interface which will accept this mock service. The user interface will have the means to add, edit, delete and show all records related to the Category including the parent. A screenshot of the user interface is shown in Figure 8-2.

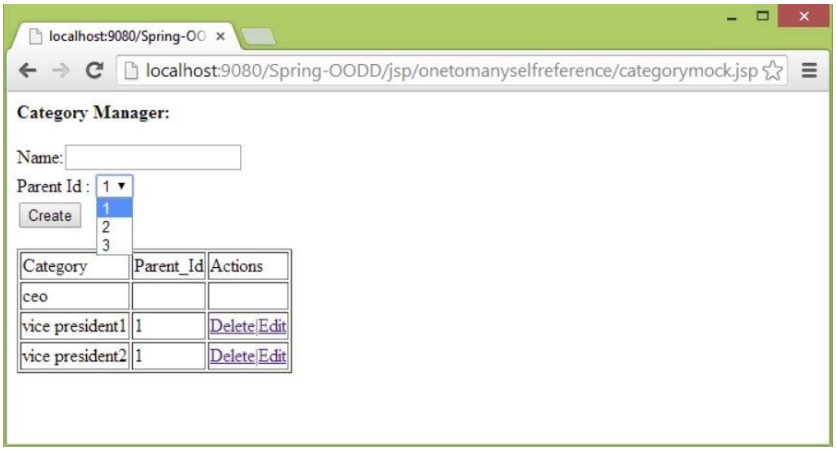

Figure 8-2: A screenshot of the Category user interface The JSP body is the inaugural page which is executed and after that the Javascript actions get called. We will see the Javascript methods one by one.

<body onload=”loadObjects()”>

Category Manager:

<form method=”post” id=”categoriesForm”>

<label>Name:<input type=”text” name=”name” id=”name”></label> <label>
<input type=”hidden” name=”Id” id=”Id”></label>

<label>Parent Id :</label>
<label id=”parentIdCombo”>
<select name=”parentId” id=”parentId”></select> </label>

<input type=submit value=”Create” id=”subButton”
onclick=”return methodCall()”>

</form>

</body>
To begin with, we will learn the JavaScript loadObjects method which is called when the page is prepared with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDonetomanyselfreference/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“name”).value=””;
alert(“Error Status Load Objects:”+textStatus); }

});
```

});

return false;

}

The processResponseData JavaScript method is called when the above service call is executed. Also, the generateTableData Javascript method is shown which generates the table structure.

```
function processResponseData(responsedata){
var dyanamicTableRow=”<table border=1>”+
“<tr>” +
“<td>Category</td>”+”<td>Parent_Id</td>”+
“<td>Actions</td>”+
“</tr>”;
var parentDropdown = “<select  id=parentId>”; var dataRow=””;
var dropdown = “”;
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue,itemno); dropdown
=dropdown+createDropDown(itemvalue);

});

dyanamicTableRow=dyanamicTableRow+dataRow+”</table>”;
if(dropdown!=””){
parentDropdown = parentDropdown+dropdown;

}

parentDropdown = parentDropdown+”</select>”;
document.getElementById(“categoryFormResponse”).innerHTML=dyanamicTal
document.getElementById(“parentIdCombo”).innerHTML=parentDropdown; }
function generateTableData(itemvalue,itemNo){
var parentId = itemvalue.parentId;
if(null == parentId) parentId=””;
var dataRow=””;
if(parentId!=””){
dataRow=”<tr>” +
“<td>” +itemvalue.name+”</td>”+
“<td>”+parentId+”</td>”+
“<td>” +
```

“Delete”+
“|Edit”+
“</td>”+
“</tr>”;
}else{
dataRow=”<tr>” +
“<td>” +itemvalue.name+”</td>”+
“<td>”+parentId+”</td>”+”<td></td>”+”</tr>”; }
return dataRow;

}

We have grabbed the page loading technique. If you detect how the JSP fragment works, you will notice a Javascript method methodCall is being called which is suitable for both create and update functionality.

```
function methodCall(){
var buttonValue = document.getElementById(“subButton”).value;
if(buttonValue==”Create”){
create();
}else if(buttonValue==”Update”){
update();

}

return false;

}
```

Next we will look at the create Javascript method which receives input from the text boxes name and parent and passes the object as a CategoryDto without an identifier.

```
function create(){
var name = $(“#name”).val();
var parentId = document.getElementById(“parentId”).value; if(parentId !=null){
parentId = parseInt(parentId);

}

var formData ={“name”:name,”parentId”:parentId}; $.ajax({
```

```
url : “Spring-OODDonetomanyselfreference/mock/create”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

document.getElementById(“name”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“name”).value=””;
document.getElementById(“parentId”).value=””; alert(“Error Status
Create:”+textStatus);

}

});

return false;

}
```

Then we will hold over the update Javascript method which keeps a CategoryDto including an identifier, name and parent.

```
function update(){
var name = $(“#name”).val();
var id = $(“#Id”).val();
var parentId = $(“#parentId”).val();
if(id==parentId){
alert(“Parent Id and Owner Id are same. Please choose a different parent.”);
return false;

}

var formData={“id”:id,”name”:name,”parentId”:parentId}; $.ajax({
url : “Spring-OODDonetomanyselfreference/mock/edit”, type: “POST”,
data : JSON.stringify(formData),
```

```
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)

{

document.getElementById("name").value="";
document.getElementById("subButton").value="Create"; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("name").value="";
alert("Error Status Update:"+textStatus);

}

});

return false;

}
```

The editObject and viewObject Javascript method calls are used to recover the selected CategoryDto object and display for editing, including the parent.

```
function editObject(Id){
editurl="Spring-OODDonetomanyselfreference/mock/findById/"+Id; var
itemForm={id:Id};
$.ajax({
url : editurl,
type: "GET",
data : itemForm,
dataType: "json",
success: function(data, textStatus, jqXHR){
viewObject(data);
document.getElementById("subButton").value="Update";
document.getElementById("Id").value=Id;
},
error: function (jqXHR, textStatus, errorThrown){
alert("Error Status Find Object:"+textStatus); }

}).
```

});

}

function viewObject(data){
document.getElementById(“name”).value=data.name; var _pId = data.parentId ;
setComboBoxValue(_pId);

}

To close the deal, we will look at the delete operation where the category identifier is passed as a response and the related Category along with parent information is removed.
function deleteObject(Id){
var productForm={id:Id};
delurl=”*Spring-OODD*onetomanyselfreference/mock/remove/”+Id; $.ajax({
url : delurl,
type: “POST”,
data : productForm,
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus);

}

});

}

Develop the Service

The succeeding step is the actual service development which involves four main steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST ability.

Develop the resource (entity) tier As an execution method, we are following the test driven development approach, hence we will first write the test for the entity and then proceed with the entity actual development. First, we create two instances of Category, one of which is the root and the other a sub-category with the root as its parent. Later, we do a reform and test whether the changes reverberated. Lastly, we remove one occurrence and test whether the count is solitary.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class CategoryTest {
@Autowired
private SessionFactory sessionFactory;
@Test
@SuppressWarnings(“unchecked”)
public void testCRUD(){
Category rootCategory = new Category();
rootCategory.setName(“Java”);
sessionFactory.getCurrentSession().save(rootCategory);
List<Category> categories =
sessionFactory.getCurrentSession().createQuery(“select category from Category
category order by id desc”).list();
Category parentCategory = categories.get(0);
Category subCategory = new Category();
subCategory.setName(“JEE”);
subCategory.setCategory(parentCategory);
sessionFactory.getCurrentSession().save(subCategory);
categories = sessionFactory.getCurrentSession().createQuery(“select category
from Category category order by id desc”).list();
Assert.assertEquals(2L, categories.size());

rootCategory = categories.get(1);
```

```
subCategory = categories.get(0);
Assert.assertEquals(rootCategory.getName(),subCategory.getCategory().getNam
subCategory.setName(“Ejb 2.1”);
subCategory.setCategory(null);
sessionFactory.getCurrentSession().merge(subCategory);
categories = sessionFactory.getCurrentSession().createQuery(“select category
from Category category order by id desc”).list(); Assert.assertEquals(null,
categories.get(0).getCategory());
sessionFactory.getCurrentSession().delete(subCategory); categories =
sessionFactory.getCurrentSession().createQuery(“select category from Category
category order by id desc”).list(); Assert.assertEquals(1L, categories.size());

                                                    }

                                                    }
```

We are developing a one-to-many selfreferencing relationship. An instance of Category may have a parent linked, except the root parent which is null. Please note that the parent can be shared with many categories with a @ManyToOne relationship.

```
@Entity
@Table(name=”CATEGORY”)
public class Category {
private Integer id;
private String name;
private Category category ;
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”id”,length=20)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }

@Column(name=”name”,length=20)
```

```
public String getName() {
return name;

                                                    }

public void setName(String name) {
this.name = name;

                                                    }

@ManyToOne(cascade=CascadeType.PERSIST)
@JoinColumn(name=”CAT_ID”,nullable=true)
public Category getCategory() {
return category;

                                                    }

public void setCategory(Category category) {
this.category = category;

                                                    }

                                                    }
```

Table structure

```
DROP TABLE `category`;
CREATE TABLE `category` (
  `id` int(11) NOT NULL AUTO_INCREMENT, `name` varchar(20)
DEFAULT NULL,
  `CAT_ID` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY (`CAT_ID`),
  CONSTRAINT FOREIGN KEY (`CAT_ID`) REFERENCES `category` (`id`)
);
```

Develop the data access tier

Next we will look at the data access tier. We will jump in with the test case for

the get all records method.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class CategoryDaoImplTest {
@Autowired
private CategoryDao dao ;
@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size());

                                                    }

                                                    }
```

We will then write the rival data access operation for the test above. The Hibernate Session.createQuery method is used with an SQL to retrieve all the records from the Category entity.

```
@Repository
@Transactional
public class CategoryDaoImpl implements CategoryDao {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Override
public List<Category> getAll() {
return sessionFactory.getCurrentSession().createQuery(“select category from
Category category order by id desc”).list(); }

                                                    }
```

Let’s now witness the test for insert operation. We insert a root category, followed by a child and then check whether the count is dual.

```
public class CategoryDaoImplTest {
@Autowired
private CategoryDao dao ;
@Test
public void testInsert(){
Category rootCategory = new Category();
```

```
rootCategory.setName(“Java”);
dao.insert(rootCategory);
Assert.assertEquals(1L, dao.getAll().size());

List<Category> categories = dao.getAll(); Category parentCategory =
categories.get(0);
Category subCategory = new Category();
subCategory.setName(“JEE”);
subCategory.setCategory(parentCategory);
dao.insert(subCategory);
Assert.assertEquals(2L, dao.getAll().size());

                                                    }

                                                    }
```

The corresponding insert operation requests the Hibernate Session.save method. The Category entity identifier gets set reliably along with its parent.

```
public class CategoryDaoImpl implements CategoryDao {
@Autowired
private SessionFactory sessionFactory;

@Override
public void insert(Category category) {
sessionFactory.getCurrentSession().save(category); }
}
```

We will now see how to write the test of finding a row with an identifier.

```
public class CategoryDaoImplTest {
@Autowired
private CategoryDao dao ;

@Test
public void testGetById() {
Category rootCategory = new Category();
rootCategory.setName(“Java”);
dao.insert(rootCategory);

List<Category> categories = dao.getAll(); Category parentCategory =
categories.get(0);
Category subCategory = new Category();
```

```
Category subCategory = new Category();
subCategory.setName(“JEE”);
subCategory.setCategory(parentCategory);
dao.insert(subCategory);

categories = dao.getAll();

Category searchCategory = dao.getById(categories.get(1).getId());
Assert.assertEquals(“Java”, searchCategory.getName()); }

                                                    }
```

The conforming data access method uses Hibernate 4 Session.get () procedure.

```
public class CategoryDaoImpl implements CategoryDao {
@Autowired
private SessionFactory sessionFactory;

@Override
public Category getById(Integer id) {
return (Category) sessionFactory.getCurrentSession().get(Category.class, id); }

                                                    }
```

We will establish the delete test procedure at this phase. First, we insert two versions of the Category followed by a delete of one of the accounts and then test whether a secluded record lasts in the stock.

```
public class CategoryDaoImplTest {
@Autowired
private CategoryDao dao ;
@Test
public void testDelete() {
Category rootCategory = new Category();
rootCategory.setName(“Java”);
dao.insert(rootCategory);

List<Category> categories = dao.getAll(); Category parentCategory =
categories.get(0);
Category subCategory = new Category();
subCategory.setName(“JEE”);
subCategory.setCategory(parentCategory);
```

```
dao.insert(subCategory);
Assert.assertEquals(2L, dao.getAll().size());

categories = dao.getAll();
Category searchCategory = categories.get(0);
searchCategory.setCategory(null);
dao.update(searchCategory);
dao.delete(searchCategory);
Assert.assertEquals(1L, dao.getAll().size());

                                                    }

                                                    }
```

The delete job uses the Hibernate 4 Session.delete () routine.

```
public class CategoryDaoImpl implements CategoryDao {
@Autowired
private SessionFactory sessionFactory;

@Override
public void delete(Category category) {
sessionFactory.getCurrentSession().delete(category); }

                                                    }
```

Finally, we will look at the update test method which inserts three records, appraises the row and tests whether the changes reverberated.

```
public class CategoryDaoImplTest {
@Test
public void testUpdate() {
Category rootCategory = new Category();
rootCategory.setName(“Java”);
dao.insert(rootCategory);

Category nextCategory = new Category();
nextCategory.setName(“Ejb 2.1”);

List<Category> categories = dao.getAll(); rootCategory = categories.get(0);
nextCategory.setCategory(rootCategory);
dao.insert(nextCategory);
```

```
categories = dao.getAll();
Category parentCategory = categories.get(0);
Category subCategory = new Category();
subCategory.setName(“JEE”);
subCategory.setCategory(parentCategory);
dao.insert(subCategory);

categories = dao.getAll();
Category searchCategory = categories.get(0);
searchCategory.setName(“Spring 3.0”);
rootCategory = categories.get(2);
searchCategory.setCategory(rootCategory);
dao.update(searchCategory);
categories = dao.getAll();


Assert.assertEquals(“Java”, categories.get(0).getCategory().getName()); }

                                                    }
```

The update data access action is obtainable below which uses Hibernate 4 Session.merge () operation.

```
public class CategoryDaoImpl implements CategoryDao {
@Autowired
private SessionFactory sessionFactory;

@Override
public void update(Category category) {
sessionFactory.getCurrentSession().merge(category); }

                                                    }
```

Develop the business service tier

We will now proceed to the business service tier. First, we will scrutinize the test for fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
```

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class CategoryServiceImplTest {
@Autowired
private CategoryService service;
@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size()); }

                                                    }
```

Let us anticipate how to package the conforming routine in the business service.

```
@Service
@Transactional
public class CategoryServiceImpl implements CategoryService{
@Autowired
private CategoryDao dao;
@Autowired
private CategoryMapper mapper;
@Override
public List<CategoryDto> findAll() {
List<Category> categories = dao.getAll(); List<CategoryDto> categoryDtos =
new ArrayList<CategoryDto>(); if(null !=categories && categories.size() > 0){
for(Category category : categories){
categoryDtos.add(mapper.mapEntityToDto(category)); }

                                                    }

return categoryDtos;

                                                    }

                                                    }
```

As articulated previously, we have a mapper for converting data access objects into entities and vice-versa, which typically has the two prominent procedures displayed below.

```
@Component
public class CategoryMapper {
```

```
public Category  mapDtoToEntity(CategoryDto categoryDto){
Category category = new Category();
if(null !=categoryDto.getId()) category.setId(categoryDto.getId()); if(null
!=categoryDto.getName()) category.setName(categoryDto.getName()); return
category;

                                                    }

public CategoryDto mapEntityToDto(Category category){
CategoryDto categoryDto = new CategoryDto();
if(null !=category.getId()) categoryDto.setId(category.getId()); if(null
!=category.getName()) categoryDto.setName(category.getName()); if(null
!=category.getCategory())
categoryDto.setParentId(category.getCategory().getId()); return categoryDto;

                                                    }

                                                    }
```

Moving on to the create test operation, which creates two incidents of the Category object, links its parent, and checks if the number of instances is the same as the fetch all records operation.

```
public class CategoryServiceImplTest {
@Autowired
private CategoryService service;
@Test
public void testCreate() {
CategoryDto parent = new CategoryDto();
parent.setName(“Java”);
service.create(parent);
Assert.assertEquals(1L, service.findAll().size());
parent = service.findAll().get(0);

CategoryDto jee = new CategoryDto();
jee.setName(“Java EE”);
jee.setParentId(parent.getId());
service.create(jee);
Assert.assertEquals(2L, service.findAll().size()); }

                                                    }
```

The compliant method in the business service tier gathers a data access object, converts to the corresponding entity via the mapper, and sends it to the data access tier for loading to the database.

```
public class CategoryServiceImpl implements CategoryService{
@Autowired
private CategoryDao dao;
@Autowired
private CategoryMapper mapper;
@Override
public void create(CategoryDto categoryDto) {
Category category = mapper.mapDtoToEntity(categoryDto);
if(null!=categoryDto.getParentId()){
Category pareCategory = dao.getById(categoryDto.getParentId());
category.setCategory(pareCategory);

                                                    }

dao.insert(category);

                                                    }

                                                    }
```

The following test operation is finding a row with a given identifier.

```
public class CategoryServiceImplTest {
@Autowired
private CategoryService service;
@Test
public void testFindById() {
CategoryDto parent = new CategoryDto();
parent.setName(“Java”);
service.create(parent);

parent = service.findAll().get(0);

CategoryDto jee = new CategoryDto();
jee.setName(“Java EE”);
jee.setParentId(parent.getId());
service.create(jee);
```

```
List<CategoryDto> parentDtos = service.findAll();
CategoryDto dto = service.findById(parentDtos.get(0).getId());
Assert.assertEquals(dto.getName(),”Java EE”);

                                                    }

                                                    }
```

The equivalent operation of business service to find a row with a given identifier would be documented now.

```
public class CategoryServiceImpl implements CategoryService{
@Autowired
private CategoryDao dao;
@Autowired
private CategoryMapper mapper;
@Override
public CategoryDto findById(Integer id) {
Category category = dao.getById(id);
if(null !=category) return mapper.mapEntityToDto(category); return null;

                                                    }

                                                    }
```

The remove test operation is next in the program. It receives a particular identifier, and eliminates the obliged row from the database. The test method first creates two models, removes one case, and checks whether there is the individual result in the database.

```
public class CategoryServiceImplTest {
@Autowired
private CategoryService service;
@Test
public void testRemove() {
CategoryDto parent = new CategoryDto();
parent.setName(“Java”);
service.create(parent);
Assert.assertEquals(1L, service.findAll().size());
parent = service.findAll().get(0);
```

```
CategoryDto jee = new CategoryDto();
jee.setName(“Java EE”);
jee.setParentId(parent.getId());
service.create(jee);
Assert.assertEquals(2L, service.findAll().size());
List<CategoryDto> parentDtos = service.findAll(); CategoryDto dto =
parentDtos.get(0);
service.remove(dto.getId());
Assert.assertEquals(1L, service.findAll().size()); }

                                                    }
```

The related business service method is recognized, which takes an identifier and removes the individual instance from the manual with the data access tier system.

```
public class CategoryServiceImpl implements CategoryService{
@Autowired
private CategoryDao dao;
@Autowired
private CategoryMapper mapper;
  @Override
public void remove(Integer id) {
Category category = dao.getById(id);
if(null != category && null != category.getCategory()){
category.setCategory(null);
dao.update(category);

                                                    }

if(null !=category) dao.delete(category);

                                                    }

                                                    }
```

The finish test operation is the edit which creates two instances, edits one of them and then checks if the edit is in effect.

```
public class CategoryServiceImplTest {
@Autowired
private CategoryService service;
```

```
@Test
public void testEdit() {
CategoryDto parent = new CategoryDto();
parent.setName(“Java”);
service.create(parent);
Assert.assertEquals(1L, service.findAll().size());
parent = service.findAll().get(0);

CategoryDto jee = new CategoryDto();
jee.setName(“Java EE”);
jee.setParentId(parent.getId());
service.create(jee);
Assert.assertEquals(2L, service.findAll().size());
List<CategoryDto> parentDtos = service.findAll(); CategoryDto dto =
parentDtos.get(0);
dto.setName(“JEE”);
service.edit(dto);



parentDtos = service.findAll();
dto = parentDtos.get(0);
Assert.assertEquals(“JEE”, dto.getName());

                                                    }

                                                    }
```

The agreeing operation of the business service is the edit operation which surrogates the call to the data access, the update method after renovation with the mapper method.

```
public class CategoryServiceImpl implements CategoryService{
@Autowired
private CategoryDao dao;
@Autowired
private CategoryMapper mapper;
@Override
public void edit(CategoryDto categoryDto) {
Category category = mapper.mapDtoToEntity(categoryDto); if(null
!=categoryDto.getParentId()){
```

```
Category parentCategory = dao.getById(categoryDto.getParentId());
category.setCategory(parentCategory);
dao.update(category);

                                                    }

                                                    }

                                                    }
```

Develop the presentation tier

The ultimate tier in the stack is the REST based Spring Controller for which we will write the test first. The test covers create, then edit, followed by a removal action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class CategoryControllerTest {
private Gson gson = new GsonBuilder().create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc =
MockMvcBuilders.<StandaloneMockMvcBuilder>webAppContextSetup
(webApplicationContext).build();

                                                    }

@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                                    }
```

}

```
public void testCreate() throws Exception {
CategoryDto categoryDto = new CategoryDto();
categoryDto.setName(“Book”);
String json = gson.toJson(categoryDto);
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“onetomanyselfreferencecreate”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}
```

```
public void testUpdate() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get(“onetomanyselfreferencefindAll”); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<CategoryDto>>() {}.getType(); List<CategoryDto>
categoryDtos = gson.fromJson(response2, listType); CategoryDto categoryDto =
categoryDtos.get(0); categoryDto.setName(“EBook”);
String json2 = gson.toJson(categoryDto);
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post(“onetomanyselfreferenceedit”);
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}
```

```
public void testDelete() throws Exception {
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get(“onetomanyselfreferencefindAll”); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<List<CategoryDto>>() {}.getType(); List<CategoryDto>
categoryDtos = gson.fromJson(response2, listType); CategoryDto categoryDto =
categoryDtos.get(0); MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post(“onetomanyselfreferenceremove/”+categoryDto.
```

```
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

                                                    }
```

The consequential REST, Spring Controller will be connecting with the business service tier which is finally over the data access tier to connect to the database. The stage is similar to the mock service, with the change that the code is working together with the genuine database and not the in memory database.

```
@Controller
@RequestMapping(value=”/onetomanyselfreference”) public class
CategoryController {
@Autowired
private CategoryService service;
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<CategoryDto> findAll(){
return service.findAll();

                                                    }

@RequestMapping(value=”/findById/{Id}”, method=RequestMethod.GET)
public @ResponseBody CategoryDto findById(@PathVariable(“Id”) Integer Id)
{
return service.findById(Id);

                                                    }

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody CategoryDto categoryDto){
service.create(categoryDto);

                                                    }

@RequestMapping(value=”/remove/{Id}”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“Id”) Integer Id){
service.remove(Id);

                                                    }
```

}

```
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody CategoryDto categoryDto){
service.edit(categoryDto);

}

}
```

The last step is the assimilation of the REST service with the unpretentious user interface. Please note that the request mapping changes from “*onetomanyselfreference*mock” to “/onetomanyselfreference.” The outstanding task includes writing the actual user interface using the mock user interface as a reference.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a one-to-many selfreferencing association
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Part IV. Managing Many-to-Many Relationships

Chapter 9. Many-to-Many Unidirectional Relationship

In this chapter, we will look in detail at how to manage a many-to-many unidirectional relationship. Working with the flow in the manuscript, we will initially look at the user interface and then the REST service package.

Domain Model

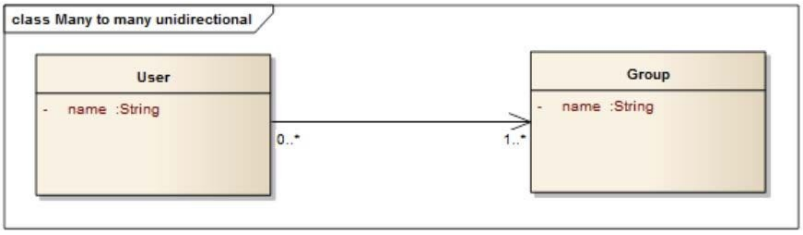

Figure 9-1: A many-to-many unidirectional relationship We have seen how to manage the different forms of one-to-many relationships in the previous chapters. In this chapter, we will view a many-to-many unidirectional association as shown in Figure 9-1. The User entity has an identifier and a name field of type String. The Group entity also has an identifier and a name String field. Every many-to-many relationship has an owner. In this example, the User is the owner. In an instance of User there must be one or more instances of Group linked, and these Group entities can in turn be shared by many User entities. Since the relationship is unidirectional you can traverse the list of Group from User but the opposite traversal is not allowed. We will follow the conventional channel of writing the mock user interface first and then the real service.

Develop a User Interface

The three vital tasks we look at during the evolution of the mock user interface are the creation of the data transfer object, the mock service and the mock user interface. We will progress one-by-one in order.

Develop the data transfer object

The UserDto data transfer object has an identifier and the name of the user. Similarly the GroupDto object has an identifier and a name. The join object UserGroupDto has an instance of UserDto and GroupDto.

```
public class UserDto {
private Integer id;
private String name;

    // getters and setters

                                        }

public class GroupDto {
private Integer id;
private String name;

    // getters and setters

                                        }

public class UserGroupDto {
private UserDto userDto;
private GroupDto groupDto;

// getters and setters

                                        }
```

Develop the mock service

We will only see how to manage the join between the User and the Group. The

usual create, read, delete and edit operations for the User as well as Group remains the same implementation, as discussed in earlier chapters. For your reference, the link to the entire source code has been given in the Appendix, should you face any issues understanding the flow. The mock service will be interacting with an in-memory database for saving the data which will be consumed by the mock user interface. We will start with the add functionality. We have a list of UserGroupDto, and with every new occurrence, one instance of UserGroupDto containing a UserDto and a GroupDto is added to the list. A Singleton design pattern by means of enum is followed for the in-memory database implementation. The identifier field is initialized with 1 and progresses by 1 for every new count.

```
public enum UserGroupInMemoryDB {

INSTANCE;
private static List<UserGroupDto> list = new ArrayList<UserGroupDto>();
public void add(UserGroupDto userGroupDto) {
list.add(userGroupDto);

                                                    }

                                                    }
```

Now let's analyze the one-and-the-same mock service implementation. The create method adds an instance of UserGroupDto from the user interface and increases the list using the mock database.

```
@Controller
@RequestMapping(value="manytomanyunidirectionalusergroup/mock") public
class UserGroupMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody UserGroupDto userGroupDto){
UserGroupInMemoryDB.INSTANCE.add(userGroupDto); }

                                                    }
```

Next we will advance to the find all rows section.

```
public enum UserGroupInMemoryDB {
public List<UserGroupDto> findAll() {
return list;
```

}

}

In the mock service, the find all rows produces a list of UserGroupDto.

```
@Controller
@RequestMapping(value=”manytomanyunidirectionalusergroup/mock”) public
class UserGroupMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<UserGroupDto> findAll(){
return UserGroupInMemoryDB.INSTANCE.findAll(); }
```

}

Next, we will implement the remove operation which eliminates a specific row with a programmed identifier.

```
public enum UserGroupInMemoryDB {
public void remove(UserGroupDto userGroupDto) {
UserGroupDto toRemove = null;
for (UserGroupDto dto:list) {
if (dto.getUserDto().getId()==userGroupDto.getUserDto().getId() &&
dto.getGroupDto().getId()==userGroupDto.getGroupDto().getId()) {
toRemove = dto;
```

}

}

```
if (toRemove!=null) list.remove(toRemove);
```

}

}

In the applicable mock service, the omit operation eradicates a specific row with the distinct identifier provided as a response to the url.

```
@Controller
@RequestMapping(value=”manytomanyunidirectionalusergroup/mock”) public
class UserGroupMockController {
@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
```

@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody UserGroupDto userGroupDto){
UserGroupInMemoryDB.INSTANCE.remove(userGroupDto); }

}

Finally, we create the isPresent operation accountable for checking if a matching row exists.
public enum UserGroupInMemoryDB {
public boolean isPresent(UserGroupDto userGroupDto) {
for (UserGroupDto dto:list) {
if (dto.getUserDto().getId() == userGroupDto.getUserDto().getId()
&& dto.getGroupDto().getId() == userGroupDto.getGroupDto().getId()) {
return true;

}

}

return false;

}

}

The isPresent operation of the mock service accepts a copy of the UserGroupDto and checks whether an existing copy of the join exists.
@Controller
@RequestMapping(value="*manytomanyunidirectional*usergroup/mock") public class UserGroupMockController {
@RequestMapping(value="/isPresent", method=RequestMethod.POST) public @ResponseBody boolean isPresent(@RequestBody UserGroupDto userGroupDto){
return UserGroupInMemoryDB.INSTANCE.isPresent(userGroupDto); }

}

Develop the mock user interface

We inspected in the prior section how to develop the mock service with an in-

memory catalog. Now we choose how to write the user interface which will consume this mock service as shown in Figure 9-2. We will be discussing the join related user interface part because the create, read, update and delete methods for the User as well as Group remains the same. If you find this confusing, please download the source code and you will see how to proceed.

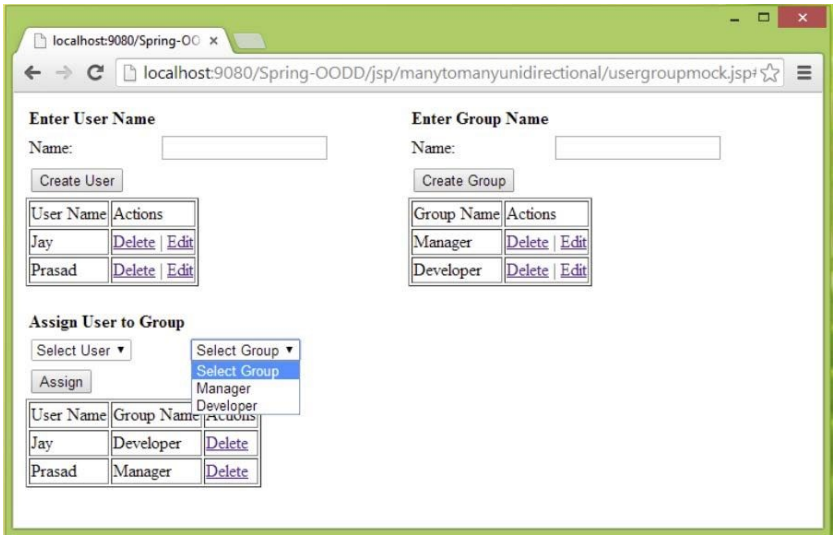

Figure 9-2: A screenshot of the User and the Group user interface The JSP body is the inaugural page which is loaded and after that the Javascript actions are called. We will draft the flow with the Javascript requests.

```
<body onload=”loadObjects()”>
<div id=”container”>
<table>
<tbody>
<tr>
<td>
<table>
<tr><td><p><b>Assign User to Group</b></p></td></tr> <tr>
<td>
<div id=”userCombo”></div>
</td>
<td>
<div id=”groupCombo”></div> </td>
</tr>
<tr>
<td><input type=submit value=”Assign” id=”subButtonUserGroup”
onclick=”return methodCall()”></td> </tr>
```

onclick="return methodCall()"></td></tr>
</table>
 </td>
</tr>

</tbody>
</table>

</body>
Initially, we will learn the JavaScript loadObjects method which is called when the page is initialized with the onload event.

```
function loadObjects(){
$.ajax({
url : "Spring-OODDmanytomanyunidirectional/user/mock/findAll", type:
"GET",
data : {},
dataType: "json",
success: function(data, textStatus, jqXHR)

                                                {

processUserResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("username").value=""; alert("Error Status Load
Objects:"+textStatus); }

                                                });

$.ajax({
url : "Spring-OODDmanytomanyunidirectional/group/mock/findAll", type:
"GET",
data : {},
dataType: "json",
success: function(data, textStatus, jqXHR)

                                                {

processGroupResponseData(data);
```

},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“groupname”).value=””; alert(“Error Status Load Objects:”+textStatus); }

});

$.ajax({
url : “*Spring-OODD*manytomanyunidirectional/usergroup/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processUserGroupResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Load Objects:”+textStatus); }

});

return false;

}

The processUserGroupResponseData JavaScript method is called when the above service call is operative. The table is printed with the resources of the in memory database. Also the generateUserGroupTableData Javascript method is shown which forms the table structure.

```
function processUserGroupResponseData(responsedata){
var dyanamicTableRow=”<table border=1>”+
“<tr>” +
“<td>User Name</td>”+”<td>Group Name</td>”+”<td>Actions</td>”+
“</tr>”;


var dataRow=””;
$.each(responsedata, function(itemno, itemvalue){
```

```
dataRow=dataRow+generateUserGroupTableData(itemvalue); });
dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("userGroupFormResponse").innerHTML=dyanamicT
}
function generateUserGroupTableData(itemvalue){
var dataRow="<tr>" +
"<td>" +itemvalue.userDto.name+"</td>"+
"<td>" +itemvalue.groupDto.name+"</td>"+
"<td>" +
"<a href=#
onclick=deleteUserGroupObject("+itemvalue.userDto.id+","+itemvalue.groupDt
"</td>"+
"</tr>";
return dataRow;
```

}

We have grasped the page loading technique. If you diagnose how the JSP fragment works, you will foresee Javascript method methodCall is being called which calls the create and update for User and Group as well as the join for User and Group.

```
function methodCall(){
var buttonValue = document.getElementById("subButtonUser").value;
if(buttonValue=="Create User"){
createUser();
} else if(buttonValue=="Update User"){
updateUser();
}
var groupButtonValue = document.getElementById("subButtonGroup").value;
if(groupButtonValue=="Create Group") {
createGroup();
} else if(groupButtonValue=="Update Group") {
updateGroup();
}
var userGroupButtonValue =
document.getElementById("subButtonUserGroup").value;
if(userGroupButtonValue=="Assign") {
isPresent();
```

}

ʃ

return false;

}

In the beginning, we will recognize the isPresent Javascript method which checks whether the join between User and Group already exists. If the relationship does not exist, a new join is formed.

```
function isPresent(){
var userid = $(“#userSelectBox”).val();
var groupid = $(“#groupSelectBox”).val();


var username = $(“#userSelectBox”).find(‘option:selected’).text(); var
groupname = $(“#groupSelectBox”).find(‘option:selected’).text();
if(null != userid && “” != userid && null!= groupid && “” != groupid) {
var formData={“userDto”:{“id”:userid},”groupDto”:{“id”:groupid}}; $.ajax({
url : “Spring-OODDmanytomanyunidirectional/usergroup/mock/isPresent”,
type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

if(!data) {
createUserGroup(userid, groupid, username, groupname); } else {
alert(“User already assigned to this group”); }
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Create:”+errorThrown);

}

});

}
```

return false;

}

We will also look at the createUserGroup Javascript method which creates the join between the selected User and Group.

```
function createUserGroup(userid,groupid, username, groupname){
var formData={“userDto”:{“id”:userid, “name”:username},”groupDto”:
{“id”:groupid, “name”:groupname}}; $.ajax({
url : “Spring-OODDmanytomanyunidirectional/usergroup/mock/create”, type:
“POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
```

{

```
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Create:”+textStatus);
```

}

});

```
return false;
```

}

Next we will scrutinize the deleteUserGroupObject Javascript method which takes the User identifier and Group identifier as an input and removes the reference from the join.

```
function deleteUserGroupObject(userid,groupid){
var formData={“userDto”:{“id”:userid},”groupDto”:{“id”:groupid}};
delurl=”Spring-OODDmanytomanyunidirectional/usergroup/mock/remove”;
$.ajax({
url : delurl,
type: “POST”,
```

```
type: "POST",
data : JSON.stringify(formData),
dataType: “json”,
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

                                                {

loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus);

                                                }

                                                });

                                                }
```

Develop the Service

The ensuing step is the actual service development which involves four main steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST capabilities. We will begin with the entity and go up to the REST service.

Develop the resource (entity) tier First, we create two instances of User and then eventually check for two instances of the user. Later, we do an update and test whether the modifications are in place. Lastly, we remove one existence and test whether the count is solitary. Also, we add two variations of Group and associate them with the remaining instances of the User.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class UserTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
public void testCRUD()

                                                {

User u1 = new User();
u1.setName(“A”);



User u2 = new User();
u2.setName(“B”);



sessionFactory.getCurrentSession().save(u1);
sessionFactory.getCurrentSession().save(u2);



u1.setName(“C”);
sessionFactory.getCurrentSession().merge(u1);

List<User> list = sessionFactory.getCurrentSession().createQuery(“from
User”).list(); Assert.assertEquals(2L, list.size());



sessionFactory.getCurrentSession().delete(u1);
List<User> list2 = sessionFactory.getCurrentSession().createQuery(“from
User”).list(); Assert.assertEquals(1L, list2.size());



Group g1 = new Group();
```

```
g1.setName(“A”);


Group g2 = new Group();
g2.setName(“B”);


sessionFactory.getCurrentSession().save(g1);
sessionFactory.getCurrentSession().save(g2);


User u3 = list2.get(0);
u3.getGroups().add(g1);
u3.getGroups().add(g2);
sessionFactory.getCurrentSession().merge(u3);


List<User> list3 = sessionFactory.getCurrentSession().createQuery(“from
User”).list(); Assert.assertEquals(1L, list3.size());
Assert.assertEquals(2L, list3.get(0).getGroups().size()); }

                                                    }
```

We will examine the Group entity which contains an identifier and a name String field. Since the entity is on the inverse side and is a unidirectional association, no reference to User is declared.

```
@Entity
@Table(name=”GROUPS”)
public class Group {

private Integer id;
private String name;

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”)
public Integer getId() {
return id;
```

}

public void setId(Integer id) {
this.id = id;

}

@Column(name=”NAME”, nullable=false, length=100) public String
getName() {
return name;

}

public void setName(String name) {
this.name = name;

}

}

We are evolving a many-to-many unidirectional association. An instance of the User may have one or more instances of a Group which can also be shared with other Users. Since the relationship is unidirectional, reference to a list of the Group is specified on the owning User side. The @ManyToMany annotation defines a many-valued association with the many-to-many multiplicity. The join table is specified on the owning side which in this case is the User. The @JoinTable annotation typically specifies the join table name along with the join column name and the inverse join column name.

```
@Entity
@Table(name = “USER”)
public class User {

private Integer id;
private String name;
private Set<Group> groups = new HashSet<Group>();
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”)
public Integer getId() {
return id;
```

```
}

public void setId(Integer id) {
this.id = id;

}

@Column(name=”NAME”, nullable=false, length=100) public String
getName() {
return name;

}

public void setName(String name) {
this.name = name;

}

@ManyToMany(cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
@JoinTable( name = “USER_GROUP”,
joinColumns = @JoinColumn(name = “USER_ID”,
referencedColumnName=”ID”),
inverseJoinColumns = @JoinColumn(name = “GROUP_ID”,
referencedColumnName=”ID”)) public Set<Group> getGroups() {
return groups;

}

public void setGroups(Set<Group> groups) {
this.groups = groups;

}

}
```

Table structure

```
DROP TABLE `user_group`;
DROP TABLE `user`;
DROP TABLE `groups`;
CREATE TABLE `user` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT
NULL,
  PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `groups` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT
NULL,
  PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `user_group` (
  `USER_ID` int(11) NOT NULL,
  `GROUP_ID` int(11) NOT NULL,
  PRIMARY KEY (`USER_ID`,`GROUP_ID`), KEY (`GROUP_ID`),
  KEY (`USER_ID`),
  CONSTRAINT FOREIGN KEY (`USER_ID`) REFERENCES `user` (`ID`),
CONSTRAINT FOREIGN KEY (`GROUP_ID`) REFERENCES `groups`
(`ID`) );
```

Develop the data access tier

Next we will look at the data access tier. We will jump to the test case for the get all records operation. Please note that we will be discussing the data access tier, which mainly manages the User and the Group join. The User data access tier along with the Group data access tier implementation remains the same as discussed in earlier chapters.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class UserGroupDaoImplTest {
```

```
public class UserGroupDaoImplTest {

@Autowired
private UserDao userDao ;

@Autowired
private GroupDao groupDao ;

@Autowired
private UserGroupDao userGroupDao;

@Test
public void testGetAll() {
Assert.assertEquals(0L, userGroupDao.getAll().size()); }

                                                    }
```

We will perceive how to write the data access operation for the test above. The Hibernate Session.createQuery method is used with an SQL to retrieve all the records from the User entity along with the Groups. The SQL retrieves a unique combination of the User and the Group join from the link table.

```
@Repository
@Transactional
public class UserGroupDaoImpl implements UserGroupDao {

@Autowired
private SessionFactory sessionFactory ;

@Override
public List<User> getAll() {
return sessionFactory.getCurrentSession().createQuery("select distinct u from
User u join u.groups g").list(); }

                                                    }
```

Let's now witness the test for isPresent operation. We insert a single row into the User entity along with Group entity and link them. Thereafter, we check if the isPresent results match this linking record.

```
public class UserGroupDaoImplTest {

@Autowired
```

```
@Autowired
private UserDao userDao ;

@Autowired
private GroupDao groupDao ;

@Autowired
private UserGroupDao userGroupDao;

@Test
public void testIsPresent() {
boolean status = false;


User u1 = new User();
u1.setName(“Alexander Mahone”);
userDao.insert(u1);


Group g1 = new Group();
g1.setName(“Java User Group”);
groupDao.insert(g1);


List<User> userList = userDao.getAll();
User user = userList.get(0);


List<Group> groupList = groupDao.getAll(); Group group = groupList.get(0);
user.getGroups().add(group);


userDao.insert(user);


List<User> userList2 = userGroupDao.isPresent(user.getId(), group.getId());
if(null != userList2) {
if(userList2.size() > 0) {
```

if(userList2.size() > 0) {
status = true;

}

}

Assert.assertTrue(status);

}

}

The matching isPresent operation just requests the Hibernate Session, createQuery method, passing the User identifier and the Group identifier and checks whether a link exists.

```
@Repository
@Transactional
public class UserGroupDaoImpl implements UserGroupDao {

@Autowired
private SessionFactory sessionFactory ;

@Override
public List<User> isPresent(Integer userid, Integer groupid) {
String hql = “select distinct u from User u join u.groups g where u.id=:userid and g.id=:groupid”; Query query =
sessionFactory.getCurrentSession().createQuery(hql);
query.setParameter(“userid”, userid);
query.setParameter(“groupid”, groupid);
return query.list();

}


}
```

Develop the business service tier

We will proceed with the business service tier. First, we will scan the test for the fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class UserGroupServiceImplTest {

@Autowired
private UserService userService;

@Autowired
private GroupService groupService;

@Autowired
private UserGroupService userGroupService;

@Test
public void testFindAll() {
Assert.assertEquals(0L, userGroupService.findAll().size()); }

                                                    }
```

Let us look at how to code the equivalent method in the business service.

```
@Service
@Transactional
public class UserGroupServiceImpl implements UserGroupService {

@Autowired
private GroupDao groupDao;

@Autowired
private UserDao userDao;

@Autowired
private UserGroupDao userGroupDao;

@Autowired
private UserMapper userMapper;

@Autowired
```

```
@Autowired
private GroupMapper groupMapper;

@Override
public List<UserGroupDto> findAll() {
List<UserGroupDto> userGroupDtos = new ArrayList<UserGroupDto>();
List<User> userList = userGroupDao.getAll(); for(User user : userList) {
UserDto userDto = userMapper.mapEntityToDto(user); Set<Group> groups =
user.getGroups();
for(Group group : groups) {
UserGroupDto userGroupDto = new UserGroupDto();
userGroupDto.setUserDto(userDto);
GroupDto groupDto = groupMapper.mapEntityToDto(group);
userGroupDto.setGroupDto(groupDto);
userGroupDtos.add(userGroupDto);

                                                    }

                                                    }

return userGroupDtos;

                                                    }

                                                    }
```

Moving on to the create test operation which creates one occurrence of the User Group link object and checks if the number of links is different from the fetch all records operation.

```
public class UserGroupServiceImplTest {

@Autowired
private UserService userService;

@Autowired
private GroupService groupService;

@Autowired
private UserGroupService userGroupService;

@T
```

```
@Test
public void testCreate() {
UserGroupDto userGroupDto = new UserGroupDto();
UserDto userDto = new UserDto();
userDto.setName(“Sara Tencradi”);
userService.create(userDto);


GroupDto groupDto = new GroupDto();
groupDto.setName(“Prison Break”);
groupService.create(groupDto);


List<UserDto> userDtos = userService.findAll(); UserDto userDto1 =
userDtos.get(0);


List<GroupDto> groupDtos = groupService.findAll(); GroupDto groupDto1 =
groupDtos.get(0);


userGroupDto.setUserDto(userDto1);
userGroupDto.setGroupDto(groupDto1);


userGroupService.create(userGroupDto);
Assert.assertEquals(1L, userGroupService.findAll().size()); }

                                                    }
```

The matching method in the business service tier collects a link data access object, fetches the unique User and Group object, and links them using the data access method.

```
public class UserGroupServiceImpl implements UserGroupService {

@Autowired
private GroupDao groupDao;
```

```
@Autowired
private UserDao userDao;

@Autowired
private UserGroupDao userGroupDao;

@Autowired
private UserMapper userMapper;

@Autowired
private GroupMapper groupMapper;

@Override
public void create(UserGroupDto userGroupDto) {
Integer userid = userGroupDto.getUserDto().getId(); Integer groupid =
userGroupDto.getGroupDto().getId();
User user = userDao.getById(userid);
Group group = groupDao.getById(groupid);
user.getGroups().add(group);



userDao.insert(user);

                                                    }

                                                    }
```

The eliminate test operation is next which creates a link between single instances of User and Group followed by a delete and then finally checks whether there is no link between the two objects.

```
public class UserGroupServiceImplTest {

@Autowired
private UserService userService;

@Autowired
private GroupService groupService;

@Autowired
private UserGroupService userGroupService;
```

```
private UserGroupService userGroupService;

@Test
public void testRemove() {
UserGroupDto userGroupDto = new UserGroupDto();
UserDto userDto = new UserDto();
userDto.setName(“Frodo Baggins”);
userService.create(userDto);


GroupDto groupDto = new GroupDto();
groupDto.setName(“The Lord of the Rings”);
groupService.create(groupDto);


List<UserDto> userDtos = userService.findAll(); UserDto userDto1 =
userDtos.get(0);


List<GroupDto> groupDtos = groupService.findAll(); GroupDto groupDto1 =
groupDtos.get(0);


userGroupDto.setUserDto(userDto1);
userGroupDto.setGroupDto(groupDto1);


userGroupService.create(userGroupDto);
Assert.assertEquals(1L, userGroupService.findAll().size());
List<UserGroupDto> uList = userGroupService.findAll(); UserGroupDto
userGroupDto1 = uList.get(0);
userGroupService.remove(userGroupDto1);


Assert.assertEquals(0L, userGroupService.findAll().size()); }

                                                    }
```

The associated business service method is drawn, which takes a linked User and Group object and removes the link from the database via the data access method.

```
public class UserGroupServiceImpl implements UserGroupService {

@Autowired
private GroupDao groupDao;

@Autowired
private UserDao userDao;

@Autowired
private UserGroupDao userGroupDao;

@Autowired
private UserMapper userMapper;

@Autowired
private GroupMapper groupMapper;

@Override
public void remove(UserGroupDto userGroupDto) {
Integer userid = userGroupDto.getUserDto().getId(); Integer groupid =
userGroupDto.getGroupDto().getId();
User user = userDao.getById(userid);
Group group = groupDao.getById(groupid);


user.getGroups().remove(group);
userDao.update(user);

                                                    }

}
```

The ending test operation is the isPresent operation which creates a link between User and Group and finally checks whether the link exists.

```
public class UserGroupServiceImplTest {

@Autowired
```

```
private UserService userService;

@Autowired
private GroupService groupService;

@Autowired
private UserGroupService userGroupService;

@Test
public void testIsPresent() {
UserGroupDto userGroupDto = new UserGroupDto();
UserDto userDto = new UserDto();
userDto.setName(“Frodo Baggins”);
userService.create(userDto);


GroupDto groupDto = new GroupDto();
groupDto.setName(“The Lord of the Rings”);
groupService.create(groupDto);


List<UserDto> userDtos = userService.findAll(); UserDto userDto1 =
userDtos.get(0);


List<GroupDto> groupDtos = groupService.findAll(); GroupDto groupDto1 =
groupDtos.get(0);


userGroupDto.setUserDto(userDto1);
userGroupDto.setGroupDto(groupDto1);


userGroupService.create(userGroupDto);
Assert.assertEquals(1L, userGroupService.findAll().size());
boolean status = userGroupService.isPresent(userGroupDto);
Assert.assertTrue(status);
```

}

}

The equivalent operation of the business service is the isPresent operation which delegates the call to the data access, the isPresent method.

```
public class UserGroupServiceImpl implements UserGroupService {

@Autowired
private GroupDao groupDao;

@Autowired
private UserDao userDao;

@Autowired
private UserGroupDao userGroupDao;

@Autowired
private UserMapper userMapper;

@Autowired
private GroupMapper groupMapper;

@Override
public boolean isPresent(UserGroupDto userGroupDto) {
boolean status = false;
List<User> userList =
userGroupDao.isPresent(userGroupDto.getUserDto().getId(),
userGroupDto.getGroupDto().getId()); if(null != userList) {
if(userList.size() > 0) {
status = true;
```

}

}

```
return status;
```

}

}

Develop the presentation tier

The finishing tier is the REST based Spring Controller for which we will view the test first. The test covers create, then checks if present followed by a delete action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class UserGroupControllerTest {

@Autowired
private UserService userService;

@Autowired
private GroupService groupService;


private Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
hh:mm:ss”).create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build(); }

@Test
public void testAll() throws Exception {
testCreate();
testPresent();
testDelete();

}
```

```
public void testCreate() throws Exception {
UserGroupDto userGroupDto = new UserGroupDto();
UserDto userDto = new UserDto();
userDto.setName(“Sara Tencradi”);
userService.create(userDto);


GroupDto groupDto = new GroupDto();
groupDto.setName(“Prison Break”);
groupService.create(groupDto);


List<UserDto> userDtos = userService.findAll(); UserDto userDto1 =
userDtos.get(0);


List<GroupDto> groupDtos = groupService.findAll(); GroupDto groupDto1 =
groupDtos.get(0);


userGroupDto.setUserDto(userDto1);
userGroupDto.setGroupDto(groupDto1);


String json = gson.toJson(userGroupDto);


MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“manytomanyunidirectionalusergroup/create”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}


public void testPresent() throws Exception {
UserGroupDto userGroupDto = new UserGroupDto();
```

List<UserDto> userDtos = userService.findAll(); UserDto userDto = userDtos.get(0);

List<GroupDto> groupDtos = groupService.findAll(); GroupDto groupDto = groupDtos.get(0);

userGroupDto.setUserDto(userDto);
userGroupDto.setGroupDto(groupDto);

String json = gson.toJson(userGroupDto);

MockHttpServletRequestBuilder requestBuilderOne = MockMvcRequestBuilders.post("*manytomanyunidirectional*usergroup/isPresent"
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}

public void testDelete() throws Exception {
UserGroupDto userGroupDto = new UserGroupDto();
List<UserDto> userDtos = userService.findAll(); UserDto userDto = userDtos.get(0);

List<GroupDto> groupDtos = groupService.findAll(); GroupDto groupDto = groupDtos.get(0);

userGroupDto.setUserDto(userDto);
userGroupDto.setGroupDto(groupDto);

```
String json = gson.toJson(userGroupDto);


MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.post("manytomanyunidirectionalusergroup/remove");
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
requestBuilder2.content(json.getBytes());
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
}

                                    }
```

The REST, Spring Controller connects with the business service tier which is ultimately contacting the data access tier to link to the database. The code is similar to the mock service, with the modification that the code is working together with the real database and not the in memory catalogue.

```
@Controller
@RequestMapping(value="manytomanyunidirectionalusergroup")
@Transactional
public class UserGroupController {

@Autowired
private UserGroupService service;


@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody List<UserGroupDto> findAll(){
return service.findAll();

                                    }

@RequestMapping(value="/isPresent", method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody UserGroupDto
userGroupDto){
return service.isPresent(userGroupDto);

                                    }

@RequestMapping(value="/create", method=RequestMethod.POST)
```

```
@ResponseBody
public void create(@RequestBody UserGroupDto userGroupDto){
service.create(userGroupDto);

                                                    }

@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody UserGroupDto userGroupDto){
service.remove(userGroupDto);

                                                    }

                                                    }
```

The concluding step is the integration of the REST service with the actual user interface. Please note that the request mapping changes from “*manytomanyunidirectional*usergroup/mock” to “*manytomanyunidirectional*usergroup”. The remaining task involves writing the real user interface using the mock user interface as a guide.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a many-to-many unidirectional association
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 10. Many-to-Many Bidirectional Relationship In this chapter, we will look in detail at how to manage a many-to-many bidirectional relationship. Going with the flow followed in the book, we will first look at the user interface and then the REST service development.

Domain Model

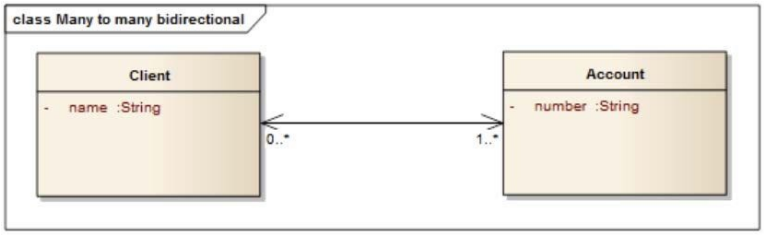

Figure 10-1: A many-to-many bidirectional relationship We will assess a many-to-many bidirectional relationship as shown in Figure 10-1. The Client entity has an identifier and a name field of type String and the Account entity is also similar. All many-to-many relationships require an owner. In this example, the Client is the owner. In an instance of Client there must be one or more instances of Account linked and these Account entities can in turn be shared by many Client entities. As the relationship is bidirectional, you can traverse the list of Accounts from Client, and the opposite traversal is also allowed. We will follow the conventional route of writing the mock user interface initially and then the factual service.

Develop a User Interface

The three main tasks we look at during the development of the mock user interface are the creation of the data transfer object, the mock service and the

mock user interface. We will go one by one in sequence.

Develop the data transfer object

The ClientDto data transfer object has an identifier and the name of the client. Similarly the AccountDto object has an identifier and a name. The join object ClientAccountDto has an instance of ClientDto and AccountDto.

```
public class ClientDto {
private Integer id;
private String name;

    // getters and setters

                                                    }

public class AccountDto {
private Integer id;
private String name;

    // getters and setters

                                                    }

public class ClientAccountDto {
private ClientDto clientDto;
private AccountDto accountDto;

// getters and setters

                                                    }
```

Develop the mock service

We will not go over the management of Client and Account objects because we have seen it many times in the previous chapters. We will only look at how to manage the link between Client and Account. If you have any issues understanding it, the entire code is shared in the Appendix section. The mock service will be fused with an in-memory custom database for saving the data

which will be used up by the mock user interface. We will start with add functionality. We have a list of ClientAccountDto, and with every new entry, one instance of ClientAccountDto having a ClientDto and an AccountDto is added to the list. A Singleton design pattern implemented as an enum is used for the in-memory database implementation. The identifier field starts with 1 and advances by 1 for every new count.

```
public enum ClientAccountInMemoryDB {

INSTANCE;

private static List<ClientAccountDto> list = new ArrayList<ClientAccountDto>
();
public void add(ClientAccountDto clientAccountDto) {
list.add(clientAccountDto);

                                                    }

                                                    }
```

Now let's evaluate the mock service implementation. The create method adds an instance of ClientAccountDto from the user interface and increases the list using the mock database.

```
@Controller
@RequestMapping(value="manytomanybidirectionalclientaccount/mock")
public class ClientAccountMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ClientAccountDto clientAccountDto){
ClientAccountInMemoryDB.INSTANCE.add(clientAccountDto); }

                                                    }
```

Next we will progress to the find all rows fragments.

```
public enum ClientAccountInMemoryDB {
public List<ClientAccountDto> findAll() {
return list;

                                                    }

                                                    }
```

In the mock service, the find all rows produces a list of ClientAccountDto.

```
public class ClientAccountMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<ClientAccountDto> findAll(){
return ClientAccountInMemoryDB.INSTANCE.findAll(); }

}
```

Next, we will implement the remove operation which reduces a specific row with an instance of ClientAccountDto containing the identifiers of Client and Account.

```
public enum ClientAccountInMemoryDB {
public void remove(ClientAccountDto clientAccountDto) {
ClientAccountDto toRemove = null;
for (ClientAccountDto dto:list) {
if (dto.getClientDto().getId()==clientAccountDto.getClientDto().getId() &&
dto.getAccountDto().getId()==clientAccountDto.getAccountDto().getId()) {
toRemove = dto;

}

}

if (toRemove!=null) list.remove(toRemove);

}

}
```

In the related mock service, the remove operation eliminates a specific row with the identifier provided as a response to the url.

```
public class ClientAccountMockController {
@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody ClientAccountDto clientAccountDto){
ClientAccountInMemoryDB.INSTANCE.remove(clientAccountDto); }

}
```

Next we create the isPresent operation responsible for testing if a matching row

exists.
public enum ClientAccountInMemoryDB {
public boolean isPresent(ClientAccountDto clientAccountDto) {
for (ClientAccountDto dto:list) {
if (dto.getClientDto().getId() == clientAccountDto.getClientDto().getId()
&& dto.getAccountDto().getId() == clientAccountDto.getAccountDto().getId())
{
return true;

}

}

return false;

}

}

The isPresent operation accepts a copy of the ClientAccountDto and checks whether a prevailing copy of the join lives.
public class ClientAccountMockController {
@RequestMapping(value=”/isPresent”, method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody ClientAccountDto
clientAccountDto){
return ClientAccountInMemoryDB.INSTANCE.isPresent(clientAccountDto); }
}

Develop the mock user interface

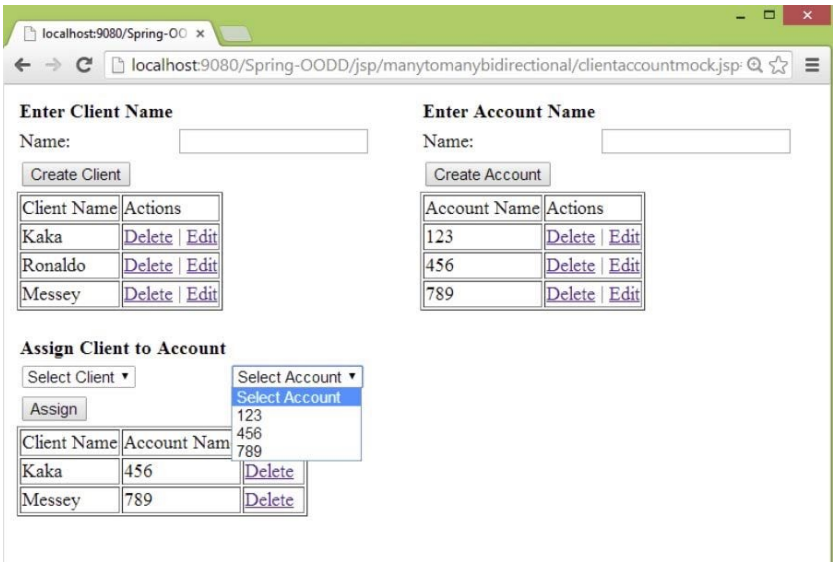

Figure 10-2: A screenshot of the Client and Account manager user interface We reviewed in the former section how to develop the mock service with an in-memory database. Now we choose how to grow the user interface shown in Figure 10-2 which will consume this mock service. We will be discussing the join related user interface part because the create, read, update and delete methods for the Client as well as Account are identical. If you have any misunderstanding, please check the source code in Appendix and you will be aware of how to progress.

The JSP body is the foundation script which becomes loaded and after that the Javascript activities are called.

```
<body onload=”loadObjects()”>
<div id=”container”>
<table>
<tbody>
<tr>
<td>
<table>
<tr><td><p><b>Assign Client to Account</b></p></td></tr> <tr>
<td>
<div id=”clientCombo”></div> </td>
<td>
<div id=”accountCombo”></div> </td>
</tr>
<tr>
<td><input type=submit value=”Assign” id=”subButtonClientAccount”
```

onclick=”return methodCall()”></td> </tr>
</table>
 </td>
</tr>
</tbody>
</table>

</body>

Initially, we will see the JavaScript loadObjects method which is called when the page is set with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDmanytomanybidirectional/client/mock/findAll”, type:
“GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processClientResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“clientname”).value=””; alert(“Error Status Load
Objects:”+textStatus); }

});

$.ajax({
url : “Spring-OODDmanytomanybidirectional/account/mock/findAll”, type:
“GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processAccountResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
```

error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“accountnumber”).value=””; alert(“Error Status Load Objects:”+textStatus); }

});

$.ajax({
url : “*Spring-OODD*manytomanybidirectional/clientaccount/mock/findAll”,
type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processClientAccountResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Load Objects:”+textStatus); }

});

return false;

}

The processClientAccountResponseData JavaScript method is called when the above service call is running. Also the generateClientAccountTableData Javascript method is shown which forms the table grid.

```
function processClientAccountResponseData(responsedata){
var dyanamicTableRow=”<table border=1>”+
“<tr>” +
“<td>Client Name</td>”+”<td>Account Name</td>”+”<td>Actions</td>”+
“</tr>”;


var dataRow=””;
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateClientAccountTableData(itemvalue); });
dyanamicTableRow=dyanamicTableRow+dataRow+”</table>”;
document.getElementById(“clientAccountFormResponse”).innerHTML=dyanam
```

```
document.getElementById("clientAccountFormResponse").innerHTML=dynaHan
}
function generateClientAccountTableData(itemvalue){
var dataRow="<tr>" +
"<td>" +itemvalue.clientDto.name+"</td>"+
"<td>" +itemvalue.accountDto.number+"</td>"+
"<td>" +
"<a href=#
onclick=deleteClientAccountObject("+itemvalue.clientDto.id+","+itemvalue.acc
"</td>"+
"</tr>";
return dataRow;

}
```

We have grasped the page loading method. If you detect how the JSP fragment works you will foresee Javascript method methodCall is being called, which calls the create and update for Client and Account as well as the link object.

```
function methodCall(){
var buttonValue = document.getElementById("subButtonClient").value;
if(buttonValue=="Create Client"){
createClient();
} else if(buttonValue=="Update Client"){
updateClient();
}


var accountButtonValue =
document.getElementById("subButtonAccount").value;
if(accountButtonValue=="Create Account") {
createAccount();
} else if(accountButtonValue=="Update Account") {
updateAccount();
}


var clientAccountButtonValue =
document.getElementById("subButtonClientAccount").value;
if(clientAccountButtonValue=="Assign") {
```

isPresent();

}

return false;

}

We will now establish the isPresent Javascript method which checks whether the join between Client and Account already exists. If the connection does not occur, a new joint is formed.

```
function isPresent(){
var clientid = $("#clientSelectBox").val();
var accountid = $("#accountSelectBox").val();


var clientname = $("#clientSelectBox").find('option:selected').text(); var
accountnumber = $("#accountSelectBox").find('option:selected').text();
if(null != clientid && "" != clientid && null!= accountid && "" != accountid) {
var formData={"clientDto":{"id":clientid},"accountDto":{"id":accountid}};
$.ajax({
url : "Spring-OODDmanytomanybidirectional/clientaccount/mock/isPresent",
type: "POST",
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)
```

{

```
if(!data) {
createClientAccount(clientid, accountid, clientname, accountnumber); } else {
alert("Client already assigned to this Account"); }
},
error: function (jqXHR, textStatus, errorThrown) {
alert("Error Status Create:"+errorThrown);
```

}

});

}

return false;

}

We will also look at the createClientAccount Javascript method which creates the join between the selected Client and Account.

```
function createClientAccount(clientid, accountid, clientname, accountnumber){
var formData={“clientDto”:{“id”:clientid, “name”:clientname},”accountDto”:
{“id”:accountid, “number”:accountnumber}}; $.ajax({
url : “Spring-OODDmanytomanybidirectional/clientaccount/mock/create”, type:
“POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
```

{

```
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Create:”+textStatus);
```

}

});

return false;

}

Next we will scrutinize the deleteClientAccountObject Javascript method which takes the Client identifier as well as the Account identifier as a response and removes the link from the join.

```
function deleteClientAccountObject(clientid,accountid){
```

```
var formData={“clientDto”:{“id”:clientid},”accountDto”:{“id”:accountid}};
delurl=”Spring-OODDmanytomanybidirectional/clientaccount/mock/remove”;
$.ajax({
url : delurl,
type: “POST”,
data : JSON.stringify(formData),
dataType: “json”,
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
                                                    {
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus);
                                                    }
                                                   });
                                                    }
```

Develop the Service

The succeeding step is the actual service development which involves four main steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST service. We will start with the entity and go up to the REST service.

Develop the resource (entity) tier First, we create two instances of Client and then, in due course, check if two instances exist. Also, well ahead, we do an update and test whether the alterations ricocheted. Next we remove

one occurrence and examine whether the count is solitary. Lastly, we add two instances of Account and associate them with the remaining instances of Client. The test results should end in two occurrences of Account related to the Client.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ClientTest {

@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
public void testCRUD() {
Client client1 = new Client();
client1.setName(«Amritendu De»);


Client client2 = new Client();
client2.setName(«Lalit Narayan Mishra»);


sessionFactory.getCurrentSession().save(client1);
sessionFactory.getCurrentSession().save(client2);
client1.setName(«Hazekul Alam»);
sessionFactory.getCurrentSession().merge(client1);
List<Client> list = sessionFactory.getCurrentSession().createQuery(“from
Client”).list(); Assert.assertEquals(2L, list.size());


sessionFactory.getCurrentSession().delete(client1);
List<Client> list2 = sessionFactory.getCurrentSession().createQuery(“from
Client”).list(); Assert.assertEquals(1L, list2.size());
```

```
Account account1 = new Account();
account1.setNumber(“Account 1”);


Account account2 = new Account();
account2.setNumber(“Account 2”);


sessionFactory.getCurrentSession().save(account1);
sessionFactory.getCurrentSession().save(account2);
Client client3 = list2.get(0);
client3.getAccounts().add(account1);
client3.getAccounts().add(account2);


sessionFactory.getCurrentSession().merge(client3);
List<Client> list3 = sessionFactory.getCurrentSession().createQuery(“from
Client”).list(); Assert.assertEquals(1L, list2.size());


Client client4 = list3.get(0);

Assert.assertEquals(2L, client4.getAccounts().size()); }

                                                    }
```

We will inspect the Client entity which comprises an identifier and a name String field. Since the entity is on the owning side and is a bidirectional association, a reference to the Account is declared. An instance of the Client may have one or more instances of Account which may also be shared with other Clients. Since the relationship is bidirectional, reference to a list of Accounts is specified on the owning Client side. The @ManyToMany annotation defines a many-valued association with many-to-many multiplicity. The join table is specified on the owning side which in this case is the Client. The @JoinTable annotation typically specifies the join table name along with the join column

name and the inverse joins column name.

```
@Entity
@Table(name=”CLIENT”)
public class Client {

private Integer id;
private String name;
private Set<Account> accounts = new HashSet<Account>();
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }

@Column(name=”NAME”, nullable=false, length=100) public String
getName() {
return name;

                                                    }

public void setName(String name) {
this.name = name;

                                                    }

@ManyToMany(cascade = CascadeType.PERSIST, fetch = FetchType.LAZY)
@JoinTable( name = “CLIENT_ACCOUNT”,
joinColumns = @JoinColumn(name = “CLIENT_ID”,
referencedColumnName=”ID”),
inverseJoinColumns = @JoinColumn(name = “ACCOUNT_ID”,
referencedColumnName=”ID”)) public Set<Account> getAccounts() {
return accounts;

                                                    }
```

}

public void setAccounts(Set<Account> accounts) {
this.accounts = accounts;

}

}

We are solving a many–to-many bidirectional relationship. If the relationship is bidirectional, the non-owning side which is Account in this case must use the mappedBy element of the @ManyToMany annotation to specify the relationship field or property of the owning side.

```
@Entity
@Table(name=”ACCOUNT”)
public class Account {


private Integer id;
private String number;
private Set<Client> clients = new HashSet<Client>();
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”)
public Integer getId() {
return id;

}

public void setId(Integer id) {
this.id = id;

}

@Column(name=”NUMBER”, nullable=false, length=100) public String
getNumber() {
return number;

}

public void setNumber(String number) {
```

```
public void setNumber(String number) {
this.number = number;

                                                    }

@ManyToMany(fetch = FetchType.LAZY, mappedBy = “accounts”) public
Set<Client> getClients() {
return clients;

                                                    }

public void setClients(Set<Client> clients) {
this.clients = clients;

                                                    }

                                                    }
```

Table structure

```
DROP TABLE `client_account`;
DROP TABLE `client`;
DROP TABLE `account`;
CREATE TABLE `client` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT
NULL,
  PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `account` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `NUMBER` varchar(100)
NOT NULL,
  PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `client_account` (
  `CLIENT_ID` int(11) NOT NULL,
  `ACCOUNT_ID` int(11) NOT NULL,
```

```
  PRIMARY KEY (`CLIENT_ID`,`ACCOUNT_ID`), KEY (`ACCOUNT_ID`),
  KEY (`CLIENT_ID`),
  CONSTRAINT FOREIGN KEY (`CLIENT_ID`) REFERENCES `client`
(`ID`), CONSTRAINT FOREIGN KEY (`ACCOUNT_ID`) REFERENCES
`account` (`ID`) );
```

Develop the data access tier

Next we will find out how to approach the data access tier. We will jump to the test case for the get all records code. Please note that we will be discussing the Client with Account link data access tier. The Client data access tier as well as the Account data access tier implementation remains identical as covered in earlier chapters.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ClientAccountDaoImplTest {

@Autowired
private ClientDao clientDao;

@Autowired
private AccountDao accountDao;

@Autowired
private ClientAccountDao clientAccountDao;

@Test
public void testGetAll() {
Assert.assertEquals(0L, clientAccountDao.getAll().size()); }

                                                    }
```

We will see how to write the matching data access operation for the test shown above. The Hibernate Session.createQuery method is used with a SQL to retrieve all the records from the Client along with the Accounts.

```
@Repository
@Transactional
public class ClientAccountDaoImpl implements ClientAccountDao {
```

```
@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings("unchecked")
@Override
public List<Client> getAll() {
return sessionFactory.getCurrentSession().createQuery("select distinct c from
Client c join c.accounts a").list(); }

                                                    }
```

Let's now see the test for the isPresent operation. We insert a single row into the Client entity along with Account entity and link them. After that we check if the isPresent result equals this joining record.

```
public class ClientAccountDaoImplTest {

@Autowired
private ClientDao clientDao;

@Autowired
private AccountDao accountDao;

@Autowired
private ClientAccountDao clientAccountDao;

@Test
public void testIsPresent() {
boolean status = false;
Client c1 = new Client();
c1.setName("Alexander Mahone");
clientDao.insert(c1);


Account a1 = new Account();
a1.setNumber("Credit Account");
accountDao.insert(a1);
```

List<Client> clientList = clientDao.getAll(); Client client = clientList.get(0);

List<Account> accountList = accountDao.getAll(); Account account = accountList.get(0);
client.getAccounts().add(account);

clientDao.insert(client);

List<Client> clientList2 = clientAccountDao.isPresent(client.getId(), account.getId()); if(null != clientList2) {
if(clientList2.size() > 0) {
status = true;

}

}

Assert.assertTrue(status);

}

}

The corresponding isPresent operation just calls the Hibernate Session.createQuery method, passing the Client identifier as well as the Account identifier, and tests whether a bond exists.
public class ClientAccountDaoImpl implements ClientAccountDao {
@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings(“unchecked”)
@Override
public List<Client> isPresent(Integer clientid, Integer accountid) {
String hql = “select distinct c from Client c join c.accounts a where c.id=:clientid and a.id=:accountid”; Query query = sessionFactory.getCurrentSession().createQuery(hql);

```
query.setParameter(“clientid”, clientid);
query.setParameter(“accountid”, accountid);
return query.list();

                                                    }

                                                    }
```

Develop the business service tier

We will then go to the business service tier. First, we will check the test for the getting all records task.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ClientAccountServiceImplTest {

@Autowired
private ClientService clientService;

@Autowired
private AccountService accountService;

@Autowired
private ClientAccountService clientAccountService;
@Test
public void testFindAll() {
Assert.assertEquals(0L, clientAccountService.findAll().size()); }

                                                    }
```

Let us review how to code the comparative method in the business service.

```
@Service
@Transactional
public class ClientAccountServiceImpl implements ClientAccountService {

@Autowired
private ClientDao clientDao;
```

```
@Autowired
private ClientMapper clientMapper;

@Autowired
private AccountDao accountDao;

@Autowired
private AccountMapper accountMapper;

@Autowired
private ClientAccountDao clientAccountDao;

@Override
public List<ClientAccountDto> findAll() {
List<ClientAccountDto> clientAccountDtos = new
ArrayList<ClientAccountDto>(); List<Client> clientList =
clientAccountDao.getAll(); for(Client client : clientList) {
ClientDto clientDto = clientMapper.mapEntityToDto(client); Set<Account>
accounts = client.getAccounts(); for(Account account : accounts) {
ClientAccountDto clientAccountDto = new ClientAccountDto();
clientAccountDto.setClientDto(clientDto);
AccountDto accountDto = accountMapper.mapEntityToDto(account);
clientAccountDto.setAccountDto(accountDto);
clientAccountDtos.add(clientAccountDto);

                                                    }

                                                    }

return clientAccountDtos;

                                                    }

                                                    }
```

Moving on to the create test operation which creates one incidence of the Client Account link object and quizzes if the number of links matches the fetch all records operation.

```
public class ClientAccountServiceImplTest {
```

```
@Autowired
private ClientService clientService;

@Autowired
private AccountService accountService;

@Autowired
private ClientAccountService clientAccountService;
@Test
public void testCreate() {
ClientAccountDto clientAccountDto = new ClientAccountDto();
ClientDto clientDto = new ClientDto();
clientDto.setName(“Sara Tencradi”);
clientService.create(clientDto);


AccountDto accountDto = new AccountDto();
accountDto.setNumber(“Savings Account”);
accountService.create(accountDto);


List<ClientDto> clientDtos = clientService.findAll(); ClientDto clientDto1 =
clientDtos.get(0);


List<AccountDto> accountDtos = accountService.findAll(); AccountDto
accountDto1 = accountDtos.get(0);


clientAccountDto.setClientDto(clientDto1);
clientAccountDto.setAccountDto(accountDto1);


clientAccountService.create(clientAccountDto); Assert.assertEquals(1L,
clientAccountService.findAll().size()); }

                                        }
```

The compliant method in the business service tier saves a linked data access object, procures the unique Client and Account object and links them via the data access method.

```
public class ClientAccountServiceImpl implements ClientAccountService {

@Autowired
private ClientDao clientDao;

@Autowired
private ClientMapper clientMapper;

@Autowired
private AccountDao accountDao;

@Autowired
private AccountMapper accountMapper;

@Autowired
private ClientAccountDao clientAccountDao;

@Override
public void create(ClientAccountDto clientAccountDto) {
Integer clientid = clientAccountDto.getClientDto().getId(); Integer accountid =
clientAccountDto.getAccountDto().getId();
Client client = clientDao.getById(clientid);
Account account = accountDao.getById(accountid);
client.getAccounts().add(account);


clientDao.insert(client);

                                                    }

                                                    }
```

The remove test operation creates a link between single instances of Client and Account followed by a delete and then checks whether there is no link between the two items.

```
public class ClientAccountServiceImplTest {
```

```
@Autowired
private ClientService clientService;

@Autowired
private AccountService accountService;

@Autowired
private ClientAccountService clientAccountService;
@Test
public void testRemove() {
ClientAccountDto clientAccountDto = new ClientAccountDto();
ClientDto clientDto = new ClientDto();
clientDto.setName(“Sara Tencradi”);
clientService.create(clientDto);


AccountDto accountDto = new AccountDto();
accountDto.setNumber(“Savings Account”);
accountService.create(accountDto);


List<ClientDto> clientDtos = clientService.findAll(); ClientDto clientDto1 =
clientDtos.get(0);


List<AccountDto> accountDtos = accountService.findAll(); AccountDto
accountDto1 = accountDtos.get(0);


clientAccountDto.setClientDto(clientDto1);
clientAccountDto.setAccountDto(accountDto1);


clientAccountService.create(clientAccountDto); Assert.assertEquals(1L,
clientAccountService.findAll().size());
List<ClientAccountDto> clientAccountList = clientAccountService.findAll();
ClientAccountDto clientAccountDto1 = clientAccountList.get(0);
```

clientAccountService.remove(clientAccountDto1);
Assert.assertEquals(0L, clientAccountService.findAll().size()); }

}

The allied business service method takes a linked Client and Account object and gets rid of the link from the database by means of the data access method.
public class ClientAccountServiceImpl implements ClientAccountService {

@Autowired
private ClientDao clientDao;

@Autowired
private ClientMapper clientMapper;

@Autowired
private AccountDao accountDao;

@Autowired
private AccountMapper accountMapper;

@Autowired
private ClientAccountDao clientAccountDao;

@Override
public void remove(ClientAccountDto clientAccountDto) {
Integer clientid = clientAccountDto.getClientDto().getId(); Integer accountid =
clientAccountDto.getAccountDto().getId();
Client client = clientDao.getById(clientid);
Account account = accountDao.getById(accountid);
client.getAccounts().remove(account);

clientDao.update(client);

}

}
The final test is the isPresent operation which creates a link between Client and

Account and to conclude checks whether there is any link between the two.

```
public class ClientAccountServiceImplTest {

@Autowired
private ClientService clientService;

@Autowired
private AccountService accountService;

@Autowired
private ClientAccountService clientAccountService;
@Test
public void testIsPresent() {
ClientAccountDto clientAccountDto = new ClientAccountDto();
ClientDto clientDto = new ClientDto();
clientDto.setName(“Sara Tencradi”);
clientService.create(clientDto);


AccountDto accountDto = new AccountDto();
accountDto.setNumber(“Savings Account”);
accountService.create(accountDto);


List<ClientDto> clientDtos = clientService.findAll(); ClientDto clientDto1 =
clientDtos.get(0);


List<AccountDto> accountDtos = accountService.findAll(); AccountDto
accountDto1 = accountDtos.get(0);


clientAccountDto.setClientDto(clientDto1);
clientAccountDto.setAccountDto(accountDto1);


clientAccountService.create(clientAccountDto); Assert.assertEquals(1L,
clientAccountService.findAll().size());
```

clientAccountService.findAll().size());
boolean status = clientAccountService.isPresent(clientAccountDto);
Assert.assertTrue(status);

}

}

The corresponding operation of the business service is the isPresent operation which delegates the call to the data access, the isPresent method.

```
public class ClientAccountServiceImpl implements ClientAccountService {

@Autowired
private ClientDao clientDao;

@Autowired
private ClientMapper clientMapper;

@Autowired
private AccountDao accountDao;

@Autowired
private AccountMapper accountMapper;

@Autowired
private ClientAccountDao clientAccountDao;

@Override
public boolean isPresent(ClientAccountDto clientAccountDto) {
boolean status = false;
List<Client> clientList =
clientAccountDao.isPresent(clientAccountDto.getClientDto().getId(),
clientAccountDto.getAccountDto().getId()); if(null != clientList) {
if(clientList.size() > 0) {
status = true;
```

}

}

return status;

return status;

}

}

Develop the presentation tier

The concluding tier in the collection is the REST based Spring Controller for which we will first follow the test. The test covers a create link object, then checks if the link is present, followed by a delete action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class ClientAccountControllerTest {
private Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
hh:mm:ss”).create();
@Autowired
private ClientService clientService;

@Autowired
private AccountService accountService;

@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc =
MockMvcBuilders.webAppContextSetup(webApplicationContext).build(); }

@Test
public void testAll() throws Exception {
testCreate();
testPresent();
testDelete();
```

}

public void testCreate() throws Exception {
ClientAccountDto clientAccountDto = new ClientAccountDto();
ClientDto clientDto = new ClientDto();
clientDto.setName(“Sara Tencradi”);
clientService.create(clientDto);

AccountDto accountDto = new AccountDto();
accountDto.setNumber(“Savings Account”);
accountService.create(accountDto);

List<ClientDto> clientDtos = clientService.findAll(); ClientDto clientDto2 =
clientDtos.get(0);

List<AccountDto> accountDtos = accountService.findAll(); AccountDto
accountDto2 = accountDtos.get(0);

clientAccountDto.setClientDto(clientDto2);
clientAccountDto.setAccountDto(accountDto2);

String json = gson.toJson(clientAccountDto);

MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“*manytomanybidirectional*clientaccount/create”)
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}

public void testPresent() throws Exception {

public void testPresent() throws Exception {
ClientAccountDto clientAccountDto = new ClientAccountDto();
List<ClientDto> clientDtos = clientService.findAll(); ClientDto clientDto =
clientDtos.get(0);

List<AccountDto> accountDtos = accountService.findAll(); AccountDto
accountDto = accountDtos.get(0);

clientAccountDto.setClientDto(clientDto);
clientAccountDto.setAccountDto(accountDto);

String json = gson.toJson(clientAccountDto);

MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post("*manytomanybidirectional*clientaccount/isPresen
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}

public void testDelete() throws Exception {
ClientAccountDto clientAccountDto = new ClientAccountDto();
List<ClientDto> clientDtos = clientService.findAll(); ClientDto clientDto =
clientDtos.get(0);

List<AccountDto> accountDtos = accountService.findAll(); AccountDto
accountDto = accountDtos.get(0);

clientAccountDto.setClientDto(clientDto);
clientAccountDto.setAccountDto(accountDto);

```
String json = gson.toJson(clientAccountDto);


MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.post("manytomanybidirectionalclientaccount/remove'
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
requestBuilder2.content(json.getBytes());
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
}

                                    }
```

The REST, Spring Controller will be linking with the business service tier which finally calls the data access tier to tie to the database. The setup is similar to the mock service, with the adjustment that the code is functioning with the real database and not the in memory set.

```
@Controller
@RequestMapping(value="manytomanybidirectionalclientaccount")
@Transactional
public class ClientAccountController {


@Autowired
private ClientAccountService service;


@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody List<ClientAccountDto> findAll(){
return service.findAll();

                                    }

@RequestMapping(value="/isPresent", method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody ClientAccountDto
clientAccountDto){
return service.isPresent(clientAccountDto);
```

```
}

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ClientAccountDto clientAccountDto){
service.create(clientAccountDto);

}

@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody ClientAccountDto clientAccountDto){
service.remove(clientAccountDto);

}

}
```

The finishing step is the incorporation of the REST service with the actual user interface. Please note that the request mapping changes from “*manytomanybidirectional*clientaccount/mock” to “*manytomanybidirectional*clientaccount”. The outstanding task involves writing the actual user interface using the mock user interface as a controller.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a many to many bidirectional associations
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 11. Many-to-Many Bidirectional with JoinAttribute Relationship

In this chapter, we will look in detail at how to manage a many-to-many bidirectional with a join attribute relationship. Going with the flow of the book, we will initially look at the user interface and then the REST package development.

Domain Model

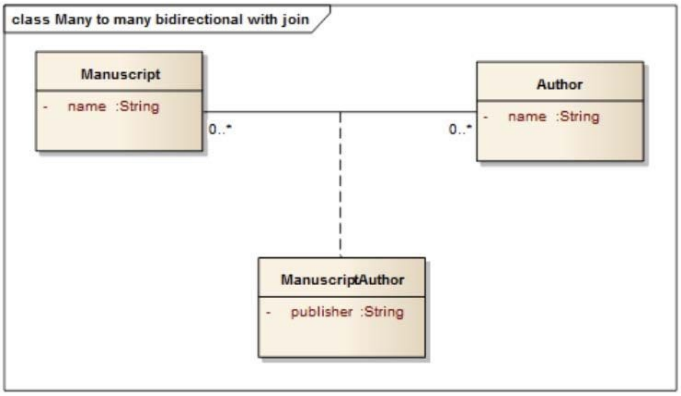

Figure 11-1: A many-to-many bidirectional with join attribute relationship We will evaluate a many-to-many bidirectional with join attribute relationship presented in Figure 11-1. The Manuscript entity has an identifier and a name field of type String and the Author entity is also similar. All many-to-many relationships necessitate an owner which is the Manuscript. The difference in this case is that the join entity has a field apart from an instance of Manuscript and Author. In an instance of Manuscript there must be one or more instances of Author linked along with the publisher and these Author entities can in turn be

shared by many Manuscript entities. As the relationship is bidirectional you can traverse the list of Authors from Manuscript and the reverse traversal is also permitted. The Manuscript and Author is associated with a publisher, hence publisher is a describing attribute of the relationship and not of the Author or the Manuscript.

Develop a User Interface

The three active tasks we study during the development of the mock user interface are the making of the data transfer object, the mock service and the mock user interface.

Develop the data transfer object

The ManuscriptDto data transfer object has an identifier and the name of the manuscript. Likewise the AuthorDto object also has an identifier and a name. The join object ManuscriptAuthorDto has an instance of ManuscriptDto and AuthorDto along with the publisher of type String.

```
public class ManuscriptDto {
private Integer id;
private String name;

    // getters and setters

                                                    }

public class AuthorDto {
private Integer id;
private String name;

    // getters and setters

                                                    }

public class ManuscriptAuthorDto {
private ManuscriptDto manuscriptDto;
private AuthorDto authorDto;
private String publisher;
```

```
// getters and setters

}
```

Develop the mock service

We will not describe the management of Manuscript and Author objects because we have seen it many times in the previous chapters. We will only see how to manage the link between Manuscript and Author. The mock service will be incorporated with an in-memory custom database for persisting the data which will be used up by the mock user interface. We will start with the add functionality. We have a list of ManuscriptAuthorDto, and with every new record, one instance of ManuscriptAuthorDto having a ManuscriptDto and an AuthorDto along with the publisher is added to the list. A Singleton design pattern by means of enum is exposed to the in-memory database application.

```
public enum ManuscriptAuthorInMemoryDB {

INSTANCE;



private static List<ManuscriptAuthorDto> list = new
ArrayList<ManuscriptAuthorDto>();
public void add(ManuscriptAuthorDto manuscriptAuthorDto) {
list.add(manuscriptAuthorDto);
}

}
```

Now let’s assess the mock service implementation. The create method adds an instance of ManuscriptAuthorDto from the user interface and increases the list using the mock database.

```
@Controller
@RequestMapping(value=”manytomanybidirectionalwithjoinattributemanuscript
public class ManuscriptAuthorMockController {
@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ManuscriptAuthorDto manuscriptAuthorDto)
{
```

{
ManuscriptAuthorInMemoryDB.INSTANCE.add(manuscriptAuthorDto); }

}

Following we will advance to the find all rows portions.

```
public enum ManuscriptAuthorInMemoryDB {
public List<ManuscriptAuthorDto> findAll() {
return list;

}

}
```

In the mock service, at this stage find all rows produces a list of ManuscriptAuthorDto.

```
public class ManuscriptAuthorMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<ManuscriptAuthorDto> findAll(){
return ManuscriptAuthorInMemoryDB.INSTANCE.findAll(); }

}
```

Next, we will devise the remove operation which reduces a specific row with a programmed identifier.

```
public enum ManuscriptAuthorInMemoryDB {
public void remove(ManuscriptAuthorDto manuscriptAuthorDto) {
ManuscriptAuthorDto toRemove = null;
for (ManuscriptAuthorDto dto:list) {
if
(dto.getManuscriptDto().getId()==manuscriptAuthorDto.getManuscriptDto().getI
&& dto.getAuthorDto().getId()==manuscriptAuthorDto.getAuthorDto().getId())
{
toRemove = dto;

}

}

if (toRemove!=null) list.remove(toRemove);

}
```

}

}

In the related mock service, the remove operation eradicates a specific row with the distinct identifier provided as an attachment to the url.

```
public class ManuscriptAuthorMockController {
@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody ManuscriptAuthorDto
manuscriptAuthorDto){
ManuscriptAuthorInMemoryDB.INSTANCE.remove(manuscriptAuthorDto); }
```

}

We then create the isPresent operation accountable for checking if a matching row exists.

```
public enum ManuscriptAuthorInMemoryDB {
public boolean isPresent(ManuscriptAuthorDto manuscriptAuthorDto) {
for (ManuscriptAuthorDto dto:list) {
if (dto.getManuscriptDto().getId() ==
manuscriptAuthorDto.getManuscriptDto().getId()
&& dto.getAuthorDto().getId() ==
manuscriptAuthorDto.getAuthorDto().getId()) {
return true;
```

}

}

```
return false;
}
```

}

The isPresent operation accepts a copy of the ManuscriptAuthorDto and checks whether an instance of the intersection exists.

```
public class ManuscriptAuthorMockController {
@RequestMapping(value=”/isPresent”, method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody ManuscriptAuthorDto
manuscriptAuthorDto){
```

```
manuscriptAuthorDto){
return
ManuscriptAuthorInMemoryDB.INSTANCE.isPresent(manuscriptAuthorDto);
}
}
```

Develop the mock user interface

We studied in the past section how to evolve the mock service with an in-memory list. Now we choose how to develop the user interface shown in Figure 11-2 which will consume this mock service. We will be discussing the join related user interface part because the create, read, update and delete methods for the Manuscript and Author are similar.

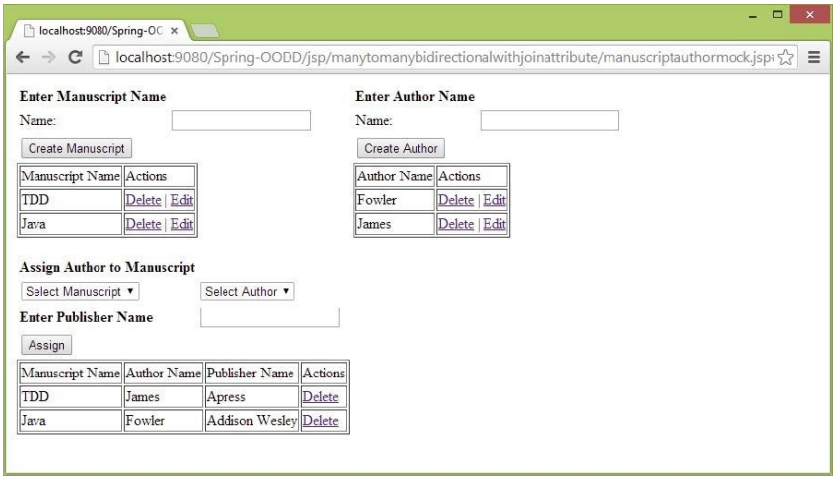

Figure 11-2: A screenshot of the Manuscript and Author manager user interface

The JSP body is the base script which is loaded and after that the Javascript events are called.

```
<body onload=”loadObjects()”>
<div id=”container”>
<table>
<tbody>
<tr>
<td>
<table>
<tr><td><p><b>Assign Author to Manuscript</b></p></td></tr> <tr>
<td>
<div id=”manuscriptCombo”></div> </td>
```

```
<td>
<div id="authorCombo"></div>
</td>
</tr>
<tr>
<td><p><b>Enter Publisher Name</b></p></td> <td>
<div id="publisherText"></div>
</td>
</tr>
<tr>
<td><input type=submit value="Assign" id="subButtonManuscriptAuthor"
onclick="return methodCall()"></td> </tr>
</table>
<div id="manuscriptAuthorFormResponse"></div> </td>
</tr>
</tbody>
</table>
</div>
</body>
```

Initially, we will view the JavaScript loadObjects method which is enabled when the page is set with the onload event.

```
function loadObjects(){
$.ajax({
url : "Spring-OODDmanytomanybidirectionalwithjoinattribute/manuscript/mock/findAll",
type: "GET",
data : {},
dataType: "json",
success: function(data, textStatus, jqXHR)

                                                {

processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown)

                                                {

document.getElementById("manuscriptname").value=""; alert("Error Status
Load Objects:"+textStatus);
```

Load Objects:"+textStatus);

}

});

$.ajax({
url : "*Spring-OODD*manytomanybidirectionalwithjoinattribute/author/mock/findAll", type: "GET",
data : {},
dataType: "json",
success: function(data, textStatus, jqXHR)

{

processAuthorResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown)

{

document.getElementById("authorname").value="";
alert("Error Status Load Objects:"+textStatus);

}

});

$.ajax({
url : "*Spring-OODD*manytomanybidirectionalwithjoinattribute/manuscriptauthor/mock/findAl
type: "GET",
data : {},
dataType: "json",
success: function(data, textStatus, jqXHR)

{

processManuscriptAuthorResponseData(data);
},

```
error: function (jqXHR, textStatus, errorThrown)

                                                {

alert("Error Status Load Objects:"+textStatus);

                                                }

                                                });

document.getElementById("publisherText").innerHTML="<input type='text'
id='publisherTextbox'/>"; return false;

                                                }
```

The processManuscriptAuthorResponseData JavaScript method is called when the above service call is operative. Also the generateClientAccountTableData Javascript method is shown which forms the table grid.

```
function processManuscriptAuthorResponseData(responsedata){
var dyanamicTableRow="<table border=1>"+
"<tr>" +
"<td>Manuscript Name</td>"+"<td>Author Name</td>"+"<td>Publisher
Name</td>"+"<td>Actions</td>"+
"</tr>";


var dataRow="";
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateManuscriptAuthorTableData(itemvalue); });
dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("manuscriptAuthorFormResponse").innerHTML=dya
}
function generateManuscriptAuthorTableData(itemvalue){
var dataRow="<tr>" +
"<td>" +itemvalue.manuscriptDto.name+"</td>"+
"<td>" +itemvalue.authorDto.name+"</td>"+
"<td>" +itemvalue.publisher+"</td>"+
"<td>" +
"<a href=#
onclick=deleteManuscriptAuthorObject("+itemvalue.manuscriptDto.id+","+item
```

```
“</td>”+
“</tr>”;
return dataRow;
```

}

We have grasped the page loading method. If you observe how the JSP fragment works, you will know Javascript method methodCall is being called which calls the create and update for Manuscript and Author as well as a link object.

```
function methodCall(){
var buttonValue = document.getElementById(“subButtonManuscript”).value;
if(buttonValue==”Create Manuscript”){
createManuscript();
} else if(buttonValue==”Update Manuscript”){
updateManuscript();
}


var authorButtonValue = document.getElementById(“subButtonAuthor”).value;
if(authorButtonValue==”Create Author”) {
createAuthor();
} else if(authorButtonValue==”Update Author”) {
updateAuthor();
}


var manuscriptAuthorButtonValue =
document.getElementById(“subButtonManuscriptAuthor”).value;
if(manuscriptAuthorButtonValue==”Assign”) {
isPresent();
```

}

```
return false;
```

}

Next, we will study the isPresent Javascript method which checks whether the join between Manuscript and Author already exists. If the linking does not

emerge, a new join is formed.

```
function isPresent(){
var manuscriptid = $("#manuscriptSelectBox").val();
var authorid = $("#authorSelectBox").val();
var publisher = $("#publisherTextbox").val();


var manuscriptname = $("#manuscriptSelectBox").find('option:selected').text();
var authorname = $("#authorSelectBox").find('option:selected').text();
if(null != manuscriptid && "" != manuscriptid && null!= authorid && "" !=
authorid && null!= publisher && "" != publisher) {
var formData={"manuscriptDto":{"id":manuscriptid},"authorDto":
{"id":authorid}}; $.ajax({
url : "Spring-
OODDmanytomanybidirectionalwithjoinattribute/manuscriptauthor/mock/isPrese
type: "POST",
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)
                                                    {

if(!data) {
createManuscriptAuthor(manuscriptid, authorid, publisher, manuscriptname,
authorname); } else {
alert("Manuscript already assigned to this Author"); }
},
error: function (jqXHR, textStatus, errorThrown)
                                                    {

alert("Error Status Create:"+errorThrown);

                                                    }

                                                   });

                                                    }
```

,

return false;

}

We will also look at the createManuscriptAuthor Javascript method which creates the join between the selected Manuscript and Author.

```
function createManuscriptAuthor(manuscriptid, authorid, publisher,
manuscriptname, authorname){
var formData={“manuscriptDto”:{“id”:manuscriptid,
“name”:manuscriptname},”authorDto”:{“id”:authorid,
“name”:authorname},”publisher”:publisher};
$.ajax({
url : “Spring-
OODDmanytomanybidirectionalwithjoinattribute/manuscriptauthor/mock/create’
type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
{
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown)
{
alert(“Error Status Create:”+textStatus);
}
});
return false;
}
```

Next we will examine the deleteManuscriptAuthorObject Javascript method which takes the Manuscript identifier as well as the Author identifier as a response and eliminates the relation from the join.

```
function deleteManuscriptAuthorObject(manuscriptid,authorid){
var formData={“manuscriptDto”:{“id”:manuscriptid},”authorDto”:
{“id”:authorid}};
delurl=”Spring-
OODDmanytomanybidirectionalwithjoinattribute/manuscriptauthor/mock/remov
$.ajax({
url : delurl,
type: “POST”,
data : JSON.stringify(formData),
dataType: “json”,
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
                                                    {
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown)
                                                    {
alert(“Error Status Delete:”+textStatus);
                                                     }
                                                    });
                                                     }
```

Develop the Service

Next is the actual service development which involves four chief steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST abilities. We will begin with the entity and go up

to the REST service.

Develop the resource (entity) tier First, we create two instances of ManuscriptAuthor and then in due course, check if two instances exist. Next, we remove one existence and examine whether the count is solitary.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class ManuscriptAuthorTest {

@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
public void testCRUD() {
Manuscript manuscript1 = new Manuscript();
manuscript1.setName(“Test Driven Application Development with Spring and
Hibernate”);
Author author1 = new Author();
author1.setName(“Amritendu De”);
sessionFactory.getCurrentSession().save(author1);


ManuscriptAuthor manuscriptAuthor1 = new ManuscriptAuthor();
manuscriptAuthor1.setManuscript(manuscript1);
manuscriptAuthor1.setAuthor(author1);
manuscriptAuthor1.setPublisher(“Createspace”);


manuscript1.getManuscriptAuthors().add(manuscriptAuthor1);
Manuscript manuscript2 = new Manuscript();
manuscript2.setName(“The Lord of the Rings”);
```

```
Author author2 = new Author(); author2.setName(“J. R. R. Tolkien”);
sessionFactory.getCurrentSession().save(author2);


ManuscriptAuthor manuscriptAuthor2 = new ManuscriptAuthor();
manuscriptAuthor2.setManuscript(manuscript2);
manuscriptAuthor2.setAuthor(author2);
manuscriptAuthor2.setPublisher(“Createspace”);
manuscript2.getManuscriptAuthors().add(manuscriptAuthor2);
sessionFactory.getCurrentSession().save(manuscript1);
sessionFactory.getCurrentSession().save(manuscript2);
author1.setName(“Amish Tripathi”);
sessionFactory.getCurrentSession().merge(author1);


List<ManuscriptAuthor> list =
sessionFactory.getCurrentSession().createQuery(“from
ManuscriptAuthor”).list(); Assert.assertEquals(2L, list.size());
List<Manuscript> manuscriptList =
sessionFactory.getCurrentSession().createQuery(“from Manuscript”).list();
Manuscript tmpManuscript = manuscriptList.get(0);
for(ManuscriptAuthor manuscriptAuthor :
tmpManuscript.getManuscriptAuthors()) {
tmpManuscript.getManuscriptAuthors().remove(manuscriptAuthor);
sessionFactory.getCurrentSession().delete(manuscriptAuthor); }
sessionFactory.getCurrentSession().merge(tmpManuscript);
List<ManuscriptAuthor> list2 =
sessionFactory.getCurrentSession().createQuery(“from
ManuscriptAuthor”).list(); Assert.assertEquals(1L, list2.size()); }
}
```

We will review the Manuscript entity which comprises an identifier and a name String field. Please note that the join entity contains an instance of Manuscript and an instance of Author together with the publisher field.

```
@Entity
@Table(name=”MANUSCRIPT”)
public class Manuscript {
```

```
private Integer id;
private String name;
private Set<ManuscriptAuthor> manuscriptAuthors = new
HashSet<ManuscriptAuthor>(0);
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name=”ID”)
public Integer getId() {
return id;

                                                    }

public void setId(Integer id) {
this.id = id;

                                                    }

@Column(name=”NAME”, nullable=false, length=100)
public String getName() {
return name;

                                                    }

public void setName(String name) {
this.name = name;

                                                    }

@OneToMany(fetch = FetchType.LAZY, mappedBy =
“manuscriptAuthorId.manuscript”, cascade=CascadeType.ALL) public
Set<ManuscriptAuthor> getManuscriptAuthors() {
return manuscriptAuthors;

                                                    }

public void setManuscriptAuthors(Set<ManuscriptAuthor> manuscriptAuthors)
{
this.manuscriptAuthors = manuscriptAuthors;

                                                    }
```

}

We are solving a many-to-many bidirectional relationship with a join attribute. If the relationship is bidirectional, the non-owning side which is Author in this case is shown below. Both the Manuscript and Author have references to ManuscriptAuthor which is the join entity having a @OneToMany relationship. In other words, the ManuscriptAuthor entity is a joining entity having a One-to-many relationship with Author and a One-to-many relationship with Manuscript, thus creating a Many-to-many relationship between Author and Manuscript.

```
@Entity
@Table(name=”AUTHOR”)
public class Author {

private Integer id;
private String name;
private Set<ManuscriptAuthor> manuscriptAuthors = new
HashSet<ManuscriptAuthor>(0);
@Id
@GeneratedValue(strategy = GenerationType.AUTO)
@Column(name=”ID”)
public Integer getId() {
return id;

}

public void setId(Integer id) {
this.id = id;

}

@Column(name=”NAME”, nullable=false, length=100)
public String getName() {
return name;

}

public void setName(String name) {
this.name = name;

}
```

```
@OneToMany(fetch = FetchType.LAZY, mappedBy =
“manuscriptAuthorId.author”) public Set<ManuscriptAuthor>
getManuscriptAuthors() {
return manuscriptAuthors;

                                                    }

public void setManuscriptAuthors(Set<ManuscriptAuthor> manuscriptAuthors)
{
this.manuscriptAuthors = manuscriptAuthors;

                                                    }

                                                    }
```

Further, we will see the ManuscriptAuthor entity containing a reference to ManuscriptAuthorId which is an @Embeddable object. Please note the use of @Transient annotation which specifies that the property is non-persistent.

```
@Entity
@Table(name = “MANUSCRIPT_AUTHOR”)
@AssociationOverrides({
      @AssociationOverride(name = “manuscriptAuthorId.manuscript”,
joinColumns = @JoinColumn(name = “MANUSCRIPT_ID”,
referencedColumnName=”ID”)), @AssociationOverride(name =
“manuscriptAuthorId.author”, joinColumns = @JoinColumn(name =
“AUTHOR_ID”, referencedColumnName=”ID”)) }) public class
ManuscriptAuthor implements Serializable {
private static final long serialVersionUID = 318229318808168400 2L;
private ManuscriptAuthorId manuscriptAuthorId = new ManuscriptAuthorId();
private String publisher;

@EmbeddedId
public ManuscriptAuthorId getManuscriptAuthorId() {
return manuscriptAuthorId;

                                                    }


public void setManuscriptAuthorId(ManuscriptAuthorId manuscriptAuthorId) {
this.manuscriptAuthorId = manuscriptAuthorId;
```

```
}

@Transient
public Manuscript getManuscript() {
return getManuscriptAuthorId().getManuscript();

}

public void setManuscript(Manuscript manuscript) {
getManuscriptAuthorId().setManuscript(manuscript);

}

@Transient
public Author getAuthor() {
return getManuscriptAuthorId().getAuthor();

}

public void setAuthor(Author author) {
getManuscriptAuthorId().setAuthor(author);

}

@Column(name=”PUBLISHER”, nullable=false, length=100) public String
getPublisher() {
return publisher;

}


public void setPublisher(String publisher) {
this.publisher = publisher;

}

}
```

The ManuscriptAuthorId is shown last to conclude the entity declarations. Note the use of @ManyToOne annotation which is on the inverse side of the

relationship.
@Embeddable
public class ManuscriptAuthorId implements java.io.Serializable {

private static final long serialVersionUID = 839588105043516589 1L;
private Manuscript manuscript;
private Author author;

@ManyToOne
public Manuscript getManuscript() {
return manuscript;

}

public void setManuscript(Manuscript manuscript) {
this.manuscript = manuscript;

}

@ManyToOne
public Author getAuthor() {
return author;

}

public void setAuthor(Author author) {
this.author = author;

}

}

Table structure

DROP TABLE `manuscript_author`;
DROP TABLE `manuscript`;
DROP TABLE `author`;
CREATE TABLE `author` (

```
   `ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT
NULL,
   PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `manuscript` (
   `ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT
NULL,
   PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `manuscript_author` (
   `PUBLISHER` varchar(100) NOT NULL,
   `MANUSCRIPT_ID` int(11) NOT NULL DEFAULT '0', `AUTHOR_ID`
int(11) NOT NULL DEFAULT '0', PRIMARY KEY (`AUTHOR_ID` ,
`MANUSCRIPT_ID`), KEY (`MANUSCRIPT_ID`),
   KEY (`AUTHOR_ID`),
   CONSTRAINT FOREIGN KEY (`AUTHOR_ID`) REFERENCES `author`
(`ID`), CONSTRAINT FOREIGN KEY (`MANUSCRIPT_ID`) REFERENCES
`manuscript` (`ID`) );
```

Develop the data access tier

Next we will review how to develop the data access tier. We will jump to the test case for the get all records operation. Please note that we will discuss the Manuscript with Author link data access tier. The Manuscript data access tier along with the Author data access tier operation remains unchanged as covered in previous chapters.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class ManuscriptAuthorDaoImplTest {

@Autowired
private AuthorDao authorDao;
```

```
@Autowired
private ManuscriptDao manuscriptDao;

@Autowired
private ManuscriptAuthorDao manuscriptAuthorDao;

@Test
public void testGetAll() {
Assert.assertEquals(0L, manuscriptAuthorDao.getAll().size()); }

                                                    }
```

We will next assess how to write the equivalent data access operation for the test shown above. The Hibernate Session.createQuery method is used with an SQL to retrieve all the records from the ManuscriptAuthor entity.

```
@Repository
@Transactional
public class ManuscriptAuthorDaoImpl implements ManuscriptAuthorDao {

@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings("unchecked")
@Override
public List<ManuscriptAuthor> getAll() {
return sessionFactory.getCurrentSession().createQuery("select distinct ma from
ManuscriptAuthor ma").list(); }

                                                    }
```

Successively, let's now see the test for the isPresent procedure. We insert a single row into the Manuscript entity along with Author entity and link them using a suitable publisher. After that we check if the isPresent returns true.

```
public class ManuscriptAuthorDaoImplTest {

@Autowired
private AuthorDao authorDao;

@Autowired
private ManuscriptDao manuscriptDao;
```

```
@Autowired
private ManuscriptAuthorDao manuscriptAuthorDao;

@Test
public void testIsPresent() {
boolean status = false;
Manuscript manuscript = new Manuscript();
manuscript.setName(“Test Driven Application Development with Spring and
Hibernate”);
Author author = new Author();
author.setName(“Amritendu De”);
authorDao.insert(author);


ManuscriptAuthor manuscriptAuthor = new ManuscriptAuthor();
manuscriptAuthor.setManuscript(manuscript);
manuscriptAuthor.setAuthor(author);
manuscriptAuthor.setPublisher(“Createspace”);


manuscript.getManuscriptAuthors().add(manuscriptAuthor);
manuscriptDao.insert(manuscript);


Manuscript manuscript2 = manuscriptDao.getAll().get(0); Author author2 =
authorDao.getAll().get(0);


List<ManuscriptAuthor> manuscriptAuthorList =
manuscriptAuthorDao.isPresent(manuscript2.getId(), author2.getId()); if(null !=
manuscriptAuthorList) {
if(manuscriptAuthorList.size() > 0) {
status = true;

                                        }

                                        }
```

```
Assert.assertTrue(status);

                                                    }

                                                    }
```

The conforming isPresent operation just requests the Hibernate Session.createQuery method, passing the Manuscript identifier as well as the Author identifier and tests whether a join exists.

```
@Repository
@Transactional
public class ManuscriptAuthorDaoImpl implements ManuscriptAuthorDao {

@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings(“unchecked”)
@Override
public List<ManuscriptAuthor> isPresent(Integer manuscriptId, Integer
authorId) {
String hql = “select distinct ma from ManuscriptAuthor ma where
ma.manuscriptAuthorId.manuscript.id=:manuscriptId and
ma.manuscriptAuthorId.author.id=:authorId”; Query query =
sessionFactory.getCurrentSession().createQuery(hql);
query.setParameter(“manuscriptId”, manuscriptId);
query.setParameter(“authorId”, authorId);
return query.list();

                                                    }

                                                    }
```

Develop the business service tier

We will shift to the business service tier. First, we will review the test for the fetching all records task.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true )
```

```
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class ManuscriptAuthorServiceImplTest {

@Autowired
private ManuscriptService manuscriptService;

@Autowired
private AuthorService authorService;

@Autowired
private ManuscriptAuthorService manuscriptAuthorService;
@Test
public void testFindAll() {
Assert.assertEquals(0L, manuscriptAuthorService.findAll().size()); }

                                                    }
```

Let us review how to code the similar method in the business service.

```
@Service
@Transactional
public class ManuscriptAuthorServiceImpl implements
ManuscriptAuthorService {

@Autowired
private ManuscriptDao manuscriptDao;

@Autowired
private ManuscriptMapper manuscriptMapper;

@Autowired
private AuthorDao authorDao;

@Autowired
private AuthorMapper authorMapper;

@Autowired
private ManuscriptAuthorDao manuscriptAuthorDao;

@Override
```

```
public List<ManuscriptAuthorDto> findAll() {
List<ManuscriptAuthorDto> manuscriptAuthorDtos = new
ArrayList<ManuscriptAuthorDto>(); List<ManuscriptAuthor> manuscriptList =
manuscriptAuthorDao.getAll(); for(ManuscriptAuthor manuscriptAuthor :
manuscriptList) {
ManuscriptAuthorDto manuscriptAuthorDto = new ManuscriptAuthorDto();
manuscriptAuthorDto.setManuscriptDto(manuscriptMapper.mapEntityToDto(ma
manuscriptAuthorDto.setAuthorDto(authorMapper.mapEntityToDto(manuscript/
manuscriptAuthorDto.setPublisher(manuscriptAuthor.getPublisher());
manuscriptAuthorDtos.add(manuscriptAuthorDto);

                                                    }

return manuscriptAuthorDtos;

                                                    }

                                                    }
```

Moving on to the create test operation which creates one incidence of the Manuscript Author link object and checks if the number of relations is equal with the fetch all records operation.

```
public class ManuscriptAuthorServiceImplTest {

@Autowired
private ManuscriptService manuscriptService;

@Autowired
private AuthorService authorService;

@Autowired
private ManuscriptAuthorService manuscriptAuthorService;
@Test
public void testCreate() {
ManuscriptAuthorDto manuscriptAuthorDto = new ManuscriptAuthorDto();
ManuscriptDto manuscriptDto = new ManuscriptDto();
manuscriptDto.setName(“The Immortals of Meluha”);
manuscriptService.create(manuscriptDto);
```

```
AuthorDto authorDto = new AuthorDto();
authorDto.setName(“Amish Tripathi”);
authorService.create(authorDto);


List<ManuscriptDto> manuscriptDtos = manuscriptService.findAll();
ManuscriptDto manuscriptDto1 = manuscriptDtos.get(0);
List<AuthorDto> authorDtos = authorService.findAll(); AuthorDto authorDto1
= authorDtos.get(0);


manuscriptAuthorDto.setManuscriptDto(manuscriptDto1);
manuscriptAuthorDto.setAuthorDto(authorDto1);
manuscriptAuthorDto.setPublisher(“Createspace”);


manuscriptAuthorService.create(manuscriptAuthorDto); Assert.assertEquals(1L,
manuscriptAuthorService.findAll().size()); }

                                                    }
```

The compliant method in the business service tier saves a linked data access object.

```
public class ManuscriptAuthorServiceImpl implements
ManuscriptAuthorService {

@Autowired
private ManuscriptDao manuscriptDao;

@Autowired
private ManuscriptMapper manuscriptMapper;

@Autowired
private AuthorDao authorDao;

@Autowired
private AuthorMapper authorMapper;

@Autowired
```

@Autowired
private ManuscriptAuthorDao manuscriptAuthorDao;

@Override
public void create(ManuscriptAuthorDto manuscriptAuthorDto) {
Integer manuscriptId = manuscriptAuthorDto.getManuscriptDto().getId();
Integer authorId = manuscriptAuthorDto.getAuthorDto().getId();
Manuscript manuscript = manuscriptDao.getById(manuscriptId); Author author = authorDao.getById(authorId);

ManuscriptAuthor manuscriptAuthor = new ManuscriptAuthor();
manuscriptAuthor.setManuscript(manuscript);
manuscriptAuthor.setAuthor(author);
manuscriptAuthor.setPublisher(manuscriptAuthorDto.getPublisher());
manuscript.getManuscriptAuthors().add(manuscriptAuthor);
manuscriptDao.insert(manuscript);

}

}

The remove test operation follows which creates a link between single instances of Manuscript and Author followed by a link delete and then notices whether there is no link between the two items.
public class ManuscriptAuthorServiceImplTest {

@Autowired
private ManuscriptService manuscriptService;

@Autowired
private AuthorService authorService;

@Autowired
private ManuscriptAuthorService manuscriptAuthorService;
@Test
public void testRemove() {
ManuscriptAuthorDto manuscriptAuthorDto = new ManuscriptAuthorDto();
ManuscriptDto manuscriptDto = new ManuscriptDto();
manuscriptDto.setName(“The Immortals of Meluha”);

manuscriptDto.setName("The Immortals of Meluha");
manuscriptService.create(manuscriptDto);

AuthorDto authorDto = new AuthorDto();
authorDto.setName("Amish Tripathi");
authorService.create(authorDto);

List<ManuscriptDto> manuscriptDtos = manuscriptService.findAll();
ManuscriptDto manuscriptDto1 = manuscriptDtos.get(0);
List<AuthorDto> authorDtos = authorService.findAll(); AuthorDto authorDto1 = authorDtos.get(0);

manuscriptAuthorDto.setManuscriptDto(manuscriptDto1);
manuscriptAuthorDto.setAuthorDto(authorDto1);
manuscriptAuthorDto.setPublisher("Createspace");

manuscriptAuthorService.create(manuscriptAuthorDto); Assert.assertEquals(1L, manuscriptAuthorService.findAll().size());
List<ManuscriptAuthorDto> manuscriptAuthorList = manuscriptAuthorService.findAll(); ManuscriptAuthorDto manuscriptAuthorDto1 = manuscriptAuthorList.get(0);
manuscriptAuthorService.remove(manuscriptAuthorDto1);
Assert.assertEquals(0, manuscriptAuthorService.findAll().size()); }

}

The connected business service method takes a linked Manuscript and Author object and removes the link from the database by means of the data access method.

```
public class ManuscriptAuthorServiceImpl implements
ManuscriptAuthorService {

@Autowired
private ManuscriptDao manuscriptDao;
```

```
@Autowired
private ManuscriptMapper manuscriptMapper;

@Autowired
private AuthorDao authorDao;

@Autowired
private AuthorMapper authorMapper;

@Autowired
private ManuscriptAuthorDao manuscriptAuthorDao;

@Override
public void remove(ManuscriptAuthorDto manuscriptAuthorDto) {
Integer manuscriptId = manuscriptAuthorDto.getManuscriptDto().getId();
Integer authorId = manuscriptAuthorDto.getAuthorDto().getId();
Manuscript manuscript = manuscriptDao.getById(manuscriptId); Author author
= authorDao.getById(authorId);


List<ManuscriptAuthor> manuscriptAuthorList =
manuscriptAuthorDao.isPresent(manuscriptId, authorId); for(ManuscriptAuthor
manuscriptAuthor : manuscriptAuthorList) {
manuscript.getManuscriptAuthors().remove(manuscriptAuthor);
author.getManuscriptAuthors().remove(manuscriptAuthor);
manuscriptAuthorDao.delete(manuscriptAuthor);

                                                    }

authorDao.update(author);
manuscriptDao.update(manuscript);

                                                    }

}
```

The ending test setup is the isPresent operation which creates a link between Manuscript and Author and then checks if there is any link between the two objects.

```
public class ManuscriptAuthorServiceImplTest {

@Autowired
private ManuscriptService manuscriptService;

@Autowired
private AuthorService authorService;

@Autowired
private ManuscriptAuthorService manuscriptAuthorService;
@Test
public void testIsPresent() {
ManuscriptAuthorDto manuscriptAuthorDto = new ManuscriptAuthorDto();
ManuscriptDto manuscriptDto = new ManuscriptDto();
manuscriptDto.setName("The Immortals of Meluha");
manuscriptService.create(manuscriptDto);


AuthorDto authorDto = new AuthorDto();
authorDto.setName("Amish Tripathi");
authorService.create(authorDto);


List<ManuscriptDto> manuscriptDtos = manuscriptService.findAll();
ManuscriptDto manuscriptDto1 = manuscriptDtos.get(0);
List<AuthorDto> authorDtos = authorService.findAll(); AuthorDto authorDto1
= authorDtos.get(0);


manuscriptAuthorDto.setManuscriptDto(manuscriptDto1);
manuscriptAuthorDto.setAuthorDto(authorDto1);
manuscriptAuthorDto.setPublisher("Createspace");


manuscriptAuthorService.create(manuscriptAuthorDto); Assert.assertEquals(1L,
manuscriptAuthorService.findAll().size());
boolean status = manuscriptAuthorService.isPresent(manuscriptAuthorDto);
Assert.assertTrue(status);
```

}

}

The analogous operation of the business service is the isPresent operation which delegates the call to the data access, the isPresent method.

```
public class ManuscriptAuthorServiceImpl implements
ManuscriptAuthorService {

@Autowired
private ManuscriptDao manuscriptDao;

@Autowired
private ManuscriptMapper manuscriptMapper;

@Autowired
private AuthorDao authorDao;

@Autowired
private AuthorMapper authorMapper;

@Autowired
private ManuscriptAuthorDao manuscriptAuthorDao;

@Override
public boolean isPresent(ManuscriptAuthorDto manuscriptAuthorDto) {
boolean status = false;
List<ManuscriptAuthor> manuscriptAuthorList =
manuscriptAuthorDao.isPresent(manuscriptAuthorDto.getManuscriptDto().getId(
manuscriptAuthorDto.getAuthorDto().getId()); if(null != manuscriptAuthorList)
{
if(manuscriptAuthorList.size() > 0) {
status = true;
```

}

}

```
return status;
```

}

}

Develop the presentation tier

The final tier in the stack is the REST based Spring Controller for which we will first do a trial test. The test creates a link object, and then checks if it is present followed by a delete action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true)
@Transactional
public class ManuscriptAuthorControllerTest {
private Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
hh:mm:ss”).create();
@Autowired
private ManuscriptService manuscriptService;

@Autowired
private AuthorService authorService;

@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc =
MockMvcBuilders.<StandaloneMockMvcBuilder>webAppContextSetup
(webApplicationContext).build();
```

}

```
@Test
public void testAll() throws Exception {
testCreate();
testPresent();
```

```
testDelete();

                                                    }

public void testCreate() throws Exception {
ManuscriptAuthorDto manuscriptAuthorDto = new ManuscriptAuthorDto();
ManuscriptDto manuscriptDto = new ManuscriptDto();
manuscriptDto.setName(“The Immortals of Meluha”);
manuscriptService.create(manuscriptDto);


AuthorDto authorDto = new AuthorDto();
authorDto.setName(“Amish Tripathi”);
authorService.create(authorDto);


List<ManuscriptDto> manuscriptDtos = manuscriptService.findAll();
ManuscriptDto manuscriptDto1 = manuscriptDtos.get(0);
List<AuthorDto> authorDtos = authorService.findAll(); AuthorDto authorDto1
= authorDtos.get(0);


manuscriptAuthorDto.setManuscriptDto(manuscriptDto1);
manuscriptAuthorDto.setAuthorDto(authorDto1);
manuscriptAuthorDto.setPublisher(“Createspace”);


String json = gson.toJson(manuscriptAuthorDto);


MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“manytomanybidirectionalwithjoinattributemanu
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}
```

```
public void testPresent() throws Exception {
ManuscriptAuthorDto manuscriptAuthorDto = new ManuscriptAuthorDto();
ManuscriptDto manuscriptDto = new ManuscriptDto();
manuscriptDto.setName(“The Immortals of Meluha”);
manuscriptService.create(manuscriptDto);


AuthorDto authorDto = new AuthorDto();
authorDto.setName(“Amish Tripathi”);
authorService.create(authorDto);


List<ManuscriptDto> manuscriptDtos = manuscriptService.findAll();
ManuscriptDto manuscriptDto1 = manuscriptDtos.get(0);
List<AuthorDto> authorDtos = authorService.findAll(); AuthorDto authorDto1
= authorDtos.get(0);


manuscriptAuthorDto.setManuscriptDto(manuscriptDto1);
manuscriptAuthorDto.setAuthorDto(authorDto1);
manuscriptAuthorDto.setPublisher(“Createspace”);


String json = gson.toJson(manuscriptAuthorDto);


MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“manytomanybidirectionalwithjoinattributemanu
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}


public void testDelete() throws Exception {
ManuscriptAuthorDto manuscriptAuthorDto = new ManuscriptAuthorDto();
ManuscriptDto manuscriptDto = new ManuscriptDto();
```

```
manuscriptDto.setName("The Immortals of Meluha");
manuscriptService.create(manuscriptDto);


AuthorDto authorDto = new AuthorDto();
authorDto.setName("Amish Tripathi");
authorService.create(authorDto);


List<ManuscriptDto> manuscriptDtos = manuscriptService.findAll();
ManuscriptDto manuscriptDto1 = manuscriptDtos.get(0);
List<AuthorDto> authorDtos = authorService.findAll(); AuthorDto authorDto1
= authorDtos.get(0);


manuscriptAuthorDto.setManuscriptDto(manuscriptDto1);
manuscriptAuthorDto.setAuthorDto(authorDto1);
manuscriptAuthorDto.setPublisher("Createspace");


String json = gson.toJson(manuscriptAuthorDto);


MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.post("manytomanybidirectionalwithjoinattributemanu
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
requestBuilder2.content(json.getBytes());
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
}

                                    }
```

The REST, Spring Controller will be interacting with the business service tier which lastly calls the data access tier to link to the database. The operation is comparable to the mock service, with the alteration that the code is working with the real database and not the in memory set.

```
@Controller
@RequestMapping(value="manytomanybidirectionalwithjoinattributemanuscript
```

```
@Transactional
public class ManuscriptAuthorController {


@Autowired
private ManuscriptAuthorService service;


@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<ManuscriptAuthorDto> findAll(){
return service.findAll();

                                                    }

@RequestMapping(value=”/isPresent”, method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody ManuscriptAuthorDto
manuscriptAuthorDto){
return service.isPresent(manuscriptAuthorDto);

                                                    }

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ManuscriptAuthorDto manuscriptAuthorDto)
{
service.create(manuscriptAuthorDto);

                                                    }

@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody ManuscriptAuthorDto
manuscriptAuthorDto){
service.remove(manuscriptAuthorDto);

                                                    }

                                                    }
```

The ultimate step is the integration of the REST service with the actual user interface. Please note that the request mapping changes from ”*manytomanybidirectionalwithjoinattribute*manuscriptauthor/mock” to “*manytomanybidirectionalwithjoinattribute*manuscriptauthor”.
The due task involves inscribing the actual user interface using the mock user interface.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a many to many bidirectional with join attribute
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 12. Many-to-Many SelfReferencing Relationship In this chapter, we will see in detail how to manage a many-to-many selfreferencing relationship. We will follow the flow projected in the book by initially looking at the user interface and then the REST package development.

Domain Model

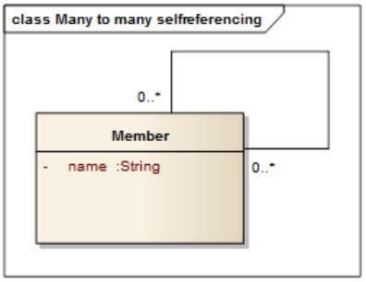

Figure 12-1: A many-to-many selfreferencing relationship We will evaluate a man-to-many selfreferencing relationship as shown in Figure 12-1. The Member entity has an identifier and a name field of type String. Here we are talking about an instance of the Member which relates to many instances of Member and they in turn are related to other Members. As the relationship is unidirectional you can traverse the list of Members from an instance of Member but you cannot traverse the reverse. An instance of Member is related to many instances of Members and hence is a selfreferencing many-to-many relationship. Please note that the relationship is of a particular type like Friends or Relatives. If the relationship changes for every instance, we have to consider the relationship covered in the next chapter. We will first look in detail at how to write the mock user interface and then the actual service.

Develop a User Interface

The three dynamic tasks we study during the development of the mock user interface are the making of the data transfer object, the mock service and the mock user interface. We will study them one by one.

Develop the data transfer object

The MemberDto data transfer object has an identifier and the name of the member. The join object MemberMemberDto has two instances of MemberDto.

```
public class MemberDto {
private Integer id;
private String name;

    // getters and setters

                                        }

public class MemberMemberDto {
private MemberDto memberId1;
private MemberDto memberId2;

// getters and setters

                                        }
```

Develop the mock service

We will not describe the management of the Member object because we have seen it many times in the preceding chapters. We will only look at how to manage the link between two instances of the Member. The mock service will be incorporated with an in-memory standard database for persisting the data which will be used by the mock user interface. We will begin to add functionality. We have a list of MemberMemberDto, and with every new record, one instance of MemberMemberDto having two instances of MemberDto is added to the list.

```
public enum MemberMemberInMemoryDB {

INSTANCE;
```

```
private static List<MemberMemberDto> list = new
ArrayList<MemberMemberDto>();
public void add(MemberMemberDto memberMemberDto) {
list.add(memberMemberDto);

                                                    }

                                                    }
```

Now let’s judge the mock service implementation. The create method adds an instance of MemberMemberDto from the user interface and increases the list using the mock database.

```
@Controller
@RequestMapping(value=”manytomanyselfreferencemembermember/mock”)
public class MemberMemberMockController {
@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody MemberMemberDto memberMemberDto){
MemberMemberInMemoryDB.INSTANCE.add(memberMemberDto); }

                                                    }
```

We will progress to the find all rows operation.

```
public enum MemberMemberInMemoryDB {
public List<MemberMemberDto> findAll() {
return list;

                                                    }

                                                    }
```

In the mock service, find all rows produces a list of MemberMemberDto.

```
public class MemberMemberMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<MemberMemberDto> findAll(){
return MemberMemberInMemoryDB.INSTANCE.findAll(); }

                                                    }
```

Next, we will consider the remove operation which lessens a precise row with a programmed identifier.

```
public enum MemberMemberInMemoryDB {
public void remove(MemberMemberDto memberMemberDto) {
MemberMemberDto toRemove = null;
for (MemberMemberDto dto:list) {
if (dto.getMemberId1().getId()==memberMemberDto.getMemberId1().getId()
&& dto.getMemberId2().getId() ==
memberMemberDto.getMemberId2().getId()) {
toRemove = dto;

                                                    }

                                                    }

if (toRemove!=null) list.remove(toRemove);

                                                    }

                                                    }
```

In the connected mock service, the remove operation eliminates a specific row with the distinct identifier provided as an attachment to the url.

```
public class MemberMemberMockController {
@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT) public void
remove(@RequestBody MemberMemberDto memberMemberDto){
MemberMemberInMemoryDB.INSTANCE.remove(memberMemberDto); }

                                                    }
```

Lastly, we create the isPresent operation responsible for analysis if a matching row exists.

```
public enum MemberMemberInMemoryDB {
public boolean isPresent(MemberMemberDto memberMemberDto) {
for (MemberMemberDto dto:list) {
if (dto.getMemberId1().getId() == memberMemberDto.getMemberId1().getId()
&& dto.getMemberId2().getId() ==
memberMemberDto.getMemberId2().getId()) {
return true;
```

} else if (dto.getMemberId1().getId() ==
memberMemberDto.getMemberId2().getId() && dto.getMemberId2().getId()
== memberMemberDto.getMemberId1().getId()) {
return true;

}

}

return false;

}

}

The isPresent operation accepts a copy of the MemberMemberDto and checks whether a copy of the join prevails.
public class MemberMemberMockController {
@RequestMapping(value="/isPresent", method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody MemberMemberDto
memberMemberDto){
return
MemberMemberInMemoryDB.INSTANCE.isPresent(memberMemberDto); }

}

Develop the mock user interface

We considered in the earlier section how to grow the mock service with an in-memory directory. Now we indicate how to sculpt the user interface shown in Figure 12-2 which will consume this mock service. We will be discussing the join related user interface part because the create, read, update and delete methods for the Member remain identical.

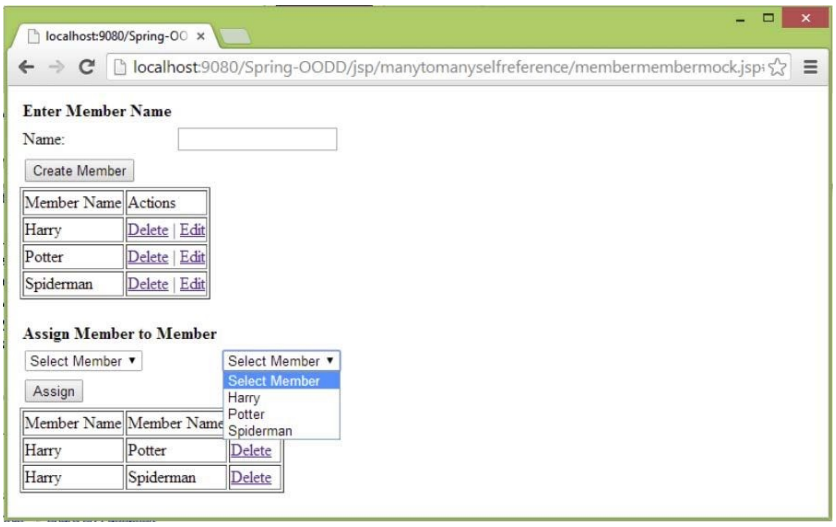

Figure 12-2: A screenshot of the Member manager user interface The JSP body is the initial code which is programmed and after that the Javascript events are called.

```
<body onload=”loadObjects()”>
<div id=”container”>
<table>
<tbody>
<tr>
<td>
<table>
<tr><td><p><b>Assign Member to Member</b></p></td></tr> <tr>
<td>
<div id=”memberCombo1”></div> </td>
<td>
<div id=”memberCombo2”></div> </td>
</tr>
<tr>
<td><input type=submit value=”Assign” id=”subButtonMemberMember”
onclick=”return methodCall()”></td> </tr>
</table>
<div id=”memberMemberFormResponse”></div> </td>
</tr>
</tbody>
</table>
</div>
</body>
```

</body>
Initially, we will accomplish the JavaScript loadObjects method which is enabled when the page is set with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDmanytomanyselfreference/member/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processResponseData(data);
processMember2ResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“membername”).value=””; alert(“Error Status Load Objects:”+textStatus); }

});

$.ajax({
url : “Spring-OODDmanytomanyselfreference/membermember/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processMemberMemberResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Load Objects:”+textStatus); }

});

return false;

}
```

The processMemberMemberResponseData JavaScript method is called when the above service call is effective. Also the generateMemberMemberTableData Javascript method is shown which processes the table grid.

```
function processMemberMemberResponseData(responsedata){
var dyanamicTableRow="<table border=1>"+
"<tr>" +
"<td>Member Name</td>"+"<td>Member Name</td>"+"<td>Actions</td>"+
"</tr>";


var dataRow="";
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateMemberMemberTableData(itemvalue); });
dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("memberMemberFormResponse").innerHTML=dyan
}
function generateMemberMemberTableData(itemvalue){
var dataRow="<tr>" +
"<td>" +itemvalue.memberId1.name+"</td>"+
"<td>" +itemvalue.memberId2.name+"</td>"+
"<td>" +
"<a href=#
onclick=deleteMemberMemberObject("+itemvalue.memberId1.id+","+itemvalue
"</td>"+
"</tr>";
return dataRow;

}
```

We have mastered the page loading method. If you see how the JSP fragment works, you will realize Javascript method methodCall is being called which calls the create and update for Member as well as a link object.

```
function methodCall(){
var buttonValue = document.getElementById("subButtonMember").value;
if(buttonValue=="Create Member"){
createMember();
} else if(buttonValue=="Update Member"){
updateMember();
```

```
}
var memberMemberButtonValue =
document.getElementById("subButtonMemberMember").value;
if(memberMemberButtonValue=="Assign") {
isPresent();

                                                    }

return false;

                                                    }
```

Next we will launch the isPresent Javascript method which checks whether the join between two instances of Member has already occurred. If the linking does not emerge, a new join is cast.

```
function isPresent(){
var memberid1 = $("#memberSelectBox1").val();
var memberid2 = $("#memberSelectBox2").val();


var membername1 = $("#memberSelectBox1").find('option:selected').text(); var
membername2 = $("#memberSelectBox2").find('option:selected').text();
if(null != memberid1 && "" != memberid1 && null!= memberid2 && "" !=
memberid2) {
var formData={"memberId1":{"id":memberid1},"memberId2":
{"id":memberid2}}; $.ajax({
url : "Spring-
OODDmanytomanyselfreference/membermember/mock/isPresent", type:
"POST",
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)

                                                    {

if(!data) {
createMemberMember(memberid1, memberid2, membername1,
membername2); } else {
```

membername2); } else {
alert(“Member already assigned to this Member”); }
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Create:”+errorThrown);

}

});

}

return false;

}

We will also look at the createMemberMember Javascript method which creates the join between the selected two instances of Member.

```
function createMemberMember(memberid1, memberid2, membername1,
membername2){
var formData={“memberId1”:{“id”:memberid1,
“name”:membername1},”memberId2”:{“id”:memberid2,
“name”:membername2}}; if(memberid1 != memberid2) {
$.ajax({
url : “Spring-OODDmanytomanyselfreference/membermember/mock/create”,
type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Create:”+textStatus);

}
```

});

} else {
alert(“Can’t assign member to itself. Please choose a different member.”); }

return false;

}

Next we will examine the deleteMemberMemberObject Javascript method which takes the two Member identifiers as a response and terminates the relationship.

```
function deleteMemberMemberObject(memberid1,memberid2){
var formData={“memberId1”:{“id”:memberid1},”memberId2”:
{“id”:memberid2}}; delurl=”Spring-
OODDmanytomanyselfreference/membermember/mock/remove”; $.ajax({
url : delurl,
type: “POST”,
data : JSON.stringify(formData),
dataType: “json”,
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
{
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Delete:”+textStatus);
}
});
}
```

Develop the Service

The succeeding step is the actual service development which includes four principal steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST competencies. We will start with the entity and go up to the REST service.

Develop the resource (entity) tier First, we create three instances of Member and then, in due course, check if three instances exist. Next, we remove one existence and examine whether the count is dual. Further, we associate an instance of Member with another and check whether the associated count is single.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class MemberTest {

@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
public void testCRUD() {
Member member1 = new Member();
member1.setName(“Amritendu De”);


Member member2 = new Member();
member2.setName(“Lalit Narayan Mishra”);


Member member3 = new Member();
member3.setName(“Hazekul Alam”);
```

```
sessionFactory.getCurrentSession().save(member1);
sessionFactory.getCurrentSession().save(member2);
sessionFactory.getCurrentSession().save(member3);
member1.setName(“Amish Tripathi”);
sessionFactory.getCurrentSession().merge(member1);
List<Member> list = sessionFactory.getCurrentSession().createQuery(“from
Member”).list(); Assert.assertEquals(3L, list.size());


sessionFactory.getCurrentSession().delete(member1);
List<Member> list2 = sessionFactory.getCurrentSession().createQuery(“from
Member”).list(); Assert.assertEquals(2L, list2.size());


member2.getMembers1().add(member3);
sessionFactory.getCurrentSession().merge(member2);
List<Member> list3 = sessionFactory.getCurrentSession().createQuery(“select
distinct m from Member m join m.members1 m1”).list();
Assert.assertEquals(1L, list3.size());


Member member4 = list3.get(0);
Assert.assertEquals(1L, member4.getMembers1().size()); }
}
```

We will review the Member entity which comprises an identifier and a name String field. Please note that the join entity contains two instances of the Member and is declared in the same class as shown below.

```
@Entity
@Table(name=”MEMBER”)
public class Member {

private Integer id;
private String name;
private Set<Member> members1 = new HashSet<Member>();
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
```

```
@Column(name=”ID”)
public Integer getId() {
return id;

                                        }

public void setId(Integer id) {
this.id = id;

                                        }

@Column(name=”NAME”, nullable=false, length=100) public String
getName() {
return name;

                                        }

public void setName(String name) {
this.name = name;

                                        }

@ManyToMany(fetch = FetchType.LAZY)
@JoinTable( name = “MEMBER_MEMBER”,
joinColumns = @JoinColumn(name = “MEMBER1_ID”,
referencedColumnName=”ID”),
inverseJoinColumns = @JoinColumn(name = “MEMBER2_ID”,
referencedColumnName=”ID”)) public Set<Member> getMembers1() {
return members1;

                                        }

public void setMembers1(Set<Member> members1) {
this.members1 = members1;

                                        }

                                        }
```

Table structure

```
DROP TABLE `member_member`;
DROP TABLE `member`;
CREATE TABLE `member` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT
NULL,
  PRIMARY KEY (`ID`)

                                                    );

CREATE TABLE `member_member` (
  `MEMBER1_ID` int(11) NOT NULL,
  `MEMBER2_ID` int(11) NOT NULL,
  PRIMARY KEY (`MEMBER1_ID`,`MEMBER2_ID`), KEY
(`MEMBER2_ID`),
  KEY (`MEMBER1_ID`),
  CONSTRAINT FOREIGN KEY (`MEMBER1_ID`) REFERENCES `member`
(`ID`), CONSTRAINT FOREIGN KEY (`MEMBER2_ID`) REFERENCES
`member` (`ID`) );
```

Develop the data access tier

Next we will look at how to write the data access tier. We will jump to the test case for the get all records programs. Please note that we will only discuss the Member over Member link data access tier. The Member data access tier remains unchanged as covered in previous chapters.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class MemberMemberDaoImplTest {

@Autowired
private MemberDao memberDao;

@Autowired
private MemberMemberDao memberMemberDao;

@Test
public void testGetAll() {
Assert.assertEquals(0L, memberMemberDao.getAll().size()); }
```

}

We will describe how to inscribe the corresponding data access operation for the test shown above. The Hibernate Session.createQuery method is used with a SQL to retrieve all the records from the Member entity.

```
@Repository
@Transactional
public class MemberMemberDaoImpl implements MemberMemberDao {

@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings("unchecked")
@Override
public List<Member> getAll() {
return sessionFactory.getCurrentSession().createQuery("select distinct m from
Member m join m.members1 m1").list(); }
```

}

Sequentially, let's now see the test for the isPresent technique. We insert two instances of Member and link them. After that we check if the isPresent method returns true, thereby confirming that the link exists.

```
public class MemberMemberDaoImplTest {

@Autowired
private MemberDao memberDao;

@Autowired
private MemberMemberDao memberMemberDao;

@Test
public void testIsPresent() {
boolean status = false;
Member m1 = new Member();
m1.setName("Alexander Mahone");
memberDao.insert(m1);
```

```
Member m2 = new Member();
m2.setName(“Credit Member”);
memberDao.insert(m2);


List<Member> memberList = memberDao.getAll(); Member member =
memberList.get(0);


List<Member> memberList2 = memberDao.getAll(); Member member2 =
memberList2.get(0);


member.getMembers1().add(member2);


memberDao.insert(member);


List<Member> memberList3 = memberMemberDao.isPresent(member.getId(),
member2.getId()); if(null != memberList3) {
if(memberList3.size() > 0) {
status = true;
                                                    }
                                                    }
Assert.assertTrue(status);
                                                    }
                                                    }
```

The imitating isPresent operation just requests the Hibernate Session.createQuery method, passing the instance identifiers, and tests whether a relationship exists.

```
public class MemberMemberDaoImpl implements MemberMemberDao {
```

```
@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings(“unchecked”)
@Override
public List<Member> isPresent(Integer memberId1, Integer memberId2) {
String hql = “select distinct m from Member m join m.members1 m1 where
m.id=:memberId1 and m1.id=:memberId2”; Query query =
sessionFactory.getCurrentSession().createQuery(hql);
query.setParameter(“memberId1”, memberId1);
query.setParameter(“memberId2”, memberId2);
return query.list();

}

}
```

Develop the business service tier

We will next review the business service tier. First, we will review the test for the fetching all records task.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class MemberMemberServiceImplTest {

@Autowired
private MemberService memberService;

@Autowired
private MemberMemberService memberMemberService;
@Test
public void testFindAll() {
Assert.assertEquals(0L, memberMemberService.findAll().size()); }

}
```

Let us examine how to program the comparative method in the business service.

```
@Service
```

```
@Service
@Transactional
public class MemberMemberServiceImpl implements MemberMemberService {

@Autowired
private MemberDao memberDao;

@Autowired
private MemberMapper memberMapper;

@Autowired
private MemberMemberDao memberMemberDao;

@Override
public List<MemberMemberDto> findAll() {
List<MemberMemberDto> memberMemberDtos = new
ArrayList<MemberMemberDto>(); List<Member> memberList =
memberMemberDao.getAll(); for(Member member : memberList) {
MemberDto memberDto = memberMapper.mapEntityToDto(member);
Set<Member> members1 = member.getMembers1(); for(Member member2 :
members1) {
MemberMemberDto memberMemberDto = new MemberMemberDto();
memberMemberDto.setMemberId1(memberDto);
MemberDto memberDto2 = memberMapper.mapEntityToDto(member2);
memberMemberDto.setMemberId2(memberDto2);
memberMemberDtos.add(memberMemberDto);

                                                    }

                                                    }

return memberMemberDtos;

                                                    }

                                                    }
```

Moving on to the create test operation which creates one occurrence of the Member link object and checks with the fetch all records operation that the link exists.

```
public class MemberMemberServiceImplTest {
```

```
public class MemberMemberServiceImplTest {

@Autowired
private MemberService memberService;

@Autowired
private MemberMemberService memberMemberService;
@Test
public void testCreate() {
MemberMemberDto memberMemberDto = new MemberMemberDto();
MemberDto memberDto1 = new MemberDto();
memberDto1.setName(“Sara Tencradi”);
memberService.create(memberDto1);


MemberDto memberDto2 = new MemberDto();
memberDto2.setName(“Mike Scofield”);
memberService.create(memberDto2);


List<MemberDto> memberDtos1 = memberService.findAll(); MemberDto
memberDto3 = memberDtos1.get(0);


List<MemberDto> memberDtos2 = memberService.findAll(); MemberDto
memberDto4 = memberDtos2.get(0);


memberMemberDto.setMemberId1(memberDto3);
memberMemberDto.setMemberId2(memberDto4);


memberMemberService.create(memberMemberDto);
Assert.assertEquals(1L, memberMemberService.findAll().size()); }

                                                    }
```

The accommodating method in the business service tier calls a Member linked

data access object.
public class MemberMemberServiceImpl implements MemberMemberService {

@Autowired
private MemberDao memberDao;

@Autowired
private MemberMapper memberMapper;

@Autowired
private MemberMemberDao memberMemberDao;

@Override
public void create(MemberMemberDto memberMemberDto) {
Integer memberid1 = memberMemberDto.getMemberId1().getId(); Integer memberid2 = memberMemberDto.getMemberId2().getId();
Member member1 = memberDao.getById(memberid1); Member member2 = memberDao.getById(memberid2); member1.getMembers1().add(member2);

memberDao.insert(member1);

}

}

The remove test task is next in line which creates a link between two instances of Member trailed by a link delete and then checks whether there is no link between the two items.
public class MemberMemberServiceImplTest {

@Autowired
private MemberService memberService;

@Autowired
private MemberMemberService memberMemberService;
@Test
public void testRemove() {
MemberMemberDto memberMemberDto = new MemberMemberDto();

```
MemberDto memberDto1 = new MemberDto();
memberDto1.setName(“Sara Tencradi”);
memberService.create(memberDto1);


MemberDto memberDto2 = new MemberDto();
memberDto2.setName(“Mike Scofield”);
memberService.create(memberDto2);


List<MemberDto> memberDtos1 = memberService.findAll(); MemberDto
memberDto3 = memberDtos1.get(0);


List<MemberDto> memberDtos2 = memberService.findAll(); MemberDto
memberDto4 = memberDtos2.get(0);


memberMemberDto.setMemberId1(memberDto3);
memberMemberDto.setMemberId2(memberDto4);


memberMemberService.create(memberMemberDto);
Assert.assertEquals(1L, memberMemberService.findAll().size());
List<MemberMemberDto> memberMemberList =
memberMemberService.findAll(); MemberMemberDto memberMemberDto1 =
memberMemberList.get(0);
memberMemberService.remove(memberMemberDto1);
Assert.assertEquals(0L, memberMemberService.findAll().size()); }

                                                    }
```

The associated business service method is revealed, which takes a linked Member object and dismisses the link from the database by means of the data access method.

```
public class MemberMemberServiceImpl implements MemberMemberService {

@Autowired
```

```
private MemberDao memberDao;

@Autowired
private MemberMapper memberMapper;

@Autowired
private MemberMemberDao memberMemberDao;

@Override
public void remove(MemberMemberDto memberMemberDto) {
Integer memberid1 = memberMemberDto.getMemberId1().getId(); Integer
memberid2 = memberMemberDto.getMemberId2().getId();
Member member1 = memberDao.getById(memberid1); Member member2 =
memberDao.getById(memberid2);
member1.getMembers1().remove(member2);
memberDao.update(member1);

                                        }

}
```

The windup test operation is the isPresent task which creates a linked Member object and then checks whether there is any link between the two objects.

```
public class MemberMemberServiceImplTest {

@Autowired
private MemberService memberService;

@Autowired
private MemberMemberService memberMemberService;
@Test
public void testIsPresent() {
MemberMemberDto memberMemberDto = new MemberMemberDto();
MemberDto memberDto1 = new MemberDto();
memberDto1.setName(“Sara Tencradi”);
memberService.create(memberDto1);


MemberDto memberDto2 = new MemberDto();
memberDto2.setName(“Mike Scofield”);
```

```
memberService.create(memberDto2);


List<MemberDto> memberDtos1 = memberService.findAll(); MemberDto
memberDto3 = memberDtos1.get(0);


List<MemberDto> memberDtos2 = memberService.findAll(); MemberDto
memberDto4 = memberDtos2.get(0);


memberMemberDto.setMemberId1(memberDto3);
memberMemberDto.setMemberId2(memberDto4);


memberMemberService.create(memberMemberDto);
Assert.assertEquals(1L, memberMemberService.findAll().size());
boolean status = memberMemberService.isPresent(memberMemberDto);
Assert.assertTrue(status);

                                                    }

                                                    }
```

The equivalent operation of the business service is the isPresent job which passes the call to the data access, the isPresent method.

```
public class MemberMemberServiceImpl implements MemberMemberService {

@Autowired
private MemberDao memberDao;

@Autowired
private MemberMapper memberMapper;

@Autowired
private MemberMemberDao memberMemberDao;

@Override
public boolean isPresent(MemberMemberDto memberMemberDto) {
```

```
public boolean isPresent(MemberMemberDto memberMemberDto) {
boolean status = false;
List<Member> memberList =
memberMemberDao.isPresent(memberMemberDto.getMemberId1().getId(),
memberMemberDto.getMemberId2().getId()); if(memberList.size() > 0) {
status = true;
} else {
memberList =
memberMemberDao.isPresent(memberMemberDto.getMemberId2().getId(),
memberMemberDto.getMemberId1().getId()); if(memberList.size() > 0) {
status = true;

                                                    }

                                                    }

return status;

                                                    }

                                                    }
```

Develop the presentation tier

The final tier in the stack is the REST based Spring Controller for which we will look at the test first. The test generates a create link object and then checks if it is present followed by a delete action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class MemberMemberControllerTest {
private Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
hh:mm:ss”).create();
@Autowired
private MemberService memberService;

@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;
```

```
@Before
public void setUp() {
mockMvc =
MockMvcBuilders.<StandaloneMockMvcBuilder>webAppContextSetup
(webApplicationContext).build();

                                                    }

@Test
public void testAll() throws Exception {
testCreate();
testPresent();
testDelete();

                                                    }

public void testCreate() throws Exception {
MemberMemberDto memberMemberDto = new MemberMemberDto();
MemberDto memberDto1 = new MemberDto();
memberDto1.setName(“Sara Tencradi”);
memberService.create(memberDto1);


MemberDto memberDto2 = new MemberDto();
memberDto2.setName(“Mike Scofield”);
memberService.create(memberDto2);


List<MemberDto> memberDtos1 = memberService.findAll(); MemberDto
memberDto3 = memberDtos1.get(0);


List<MemberDto> memberDtos2 = memberService.findAll(); MemberDto
memberDto4 = memberDtos2.get(0);


memberMemberDto.setMemberId1(memberDto3);
```

memberMemberDto.setMemberId2(memberDto4);

String json = gson.toJson(memberMemberDto);

MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders. post (“*manytomanyselfreference*member
member/create”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}

public void testPresent() throws Exception {
MemberMemberDto memberMemberDto = new MemberMemberDto();
List<MemberDto> memberDtos1 = memberService.findAll(); MemberDto
memberDto1 = memberDtos1.get(0);

List<MemberDto> memberDtos2 = memberService.findAll(); MemberDto
memberDto2 = memberDtos2.get(0);

memberMemberDto.setMemberId1(memberDto1);
memberMemberDto.setMemberId2(memberDto2);

String json = gson.toJson(memberMemberDto);

MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.*post*(“*manytomanyselfreference*membermember/isPre
requestBuilderOne.contentType(MediaType.*APPLICATION_JSON*);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers

```
}

public void testDelete() throws Exception {
MemberMemberDto memberMemberDto = new MemberMemberDto();
List<MemberDto> memberDtos1 = memberService.findAll(); MemberDto
memberDto1 = memberDtos1.get(0);

List<MemberDto> memberDtos2 = memberService.findAll(); MemberDto
memberDto2 = memberDtos2.get(0);

memberMemberDto.setMemberId1(memberDto1);
memberMemberDto.setMemberId2(memberDto2);

String json = gson.toJson(memberMemberDto);

MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.post("manytomanyselfreferencemembermember/remo
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
requestBuilder2.content(json.getBytes());
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
}

                                    }
```

The REST, Spring Controller will be interacting with the business service tier which lastly requests the data access tier to link to the database. The setup is similar to the mock service, with the modification that the code is working with the genuine database and not the in memory register.

```
@Controller
@RequestMapping(value="manytomanyselfreferencemembermember")
@Transactional
public class MemberMemberController {
```

```
@Autowired
private MemberMemberService service;


@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<MemberMemberDto> findAll(){
return service.findAll();

                                                    }

@RequestMapping(value=”/isPresent”, method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody MemberMemberDto
memberMemberDto){
return service.isPresent(memberMemberDto);

                                                    }

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody MemberMemberDto memberMemberDto){
service.create(memberMemberDto);

                                                    }

@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT) public void
remove(@RequestBody MemberMemberDto memberMemberDto){
service.remove(memberMemberDto);

                                                    }

                                                    }
```

The decisive step in the development is the mixing of the REST service with the actual user interface. Please note that the request mapping changes from “*manytomanyselfreference*membermember/mock” to “*manytomanyselfreference*membermember.”

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a many-to-many selfreferencing relationship
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 13. Many-to-Many SelfReferencing with JoinAttribute Relationship

In this chapter, we will elaborate how to manage a many-to-many selfreferencing with joinattribute relationship. We will follow the stream developed in the book by first looking at the user interface and then the REST service development.

Domain Model

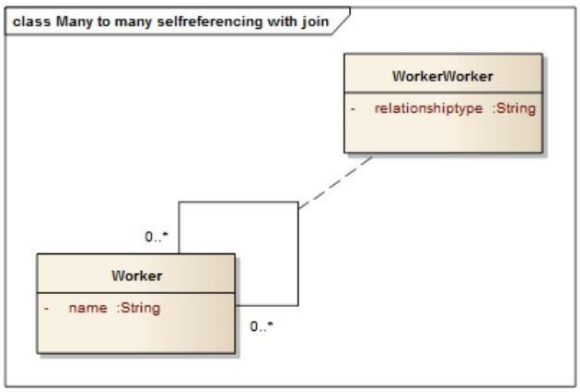

Figure 13-1: A many-to-many selfreferencing with join attribute relationship We will evaluate a many-to-many selfreferencing with join attribute relationship as illustrated in Figure 13-1. The Worker entity has an identifier and a name field of type String. Here we are speaking about an instance of the Worker which relates to many instances of Worker with a relationship type of String field, and they in turn are related to other Workers. The relationship type attribute is used to indicate the relationship between the two Workers as a Father, Sister or Brother, for example. As the relationship is unidirectional you can traverse the

list of Workers from an instance of Worker but you cannot navigate the contrary. We will walk through the traditional way of writing the mock user interface first and then the genuine service.

Develop a User Interface

The three vigorous tasks we learn during the evolution of the mock user interface are the development of the data transfer object, the mock service and the mock user interface. We develop them one by one.

Develop the data transfer object

The WorkerDto data transfer object has an identifier and the name of the worker. The join object WorkerWorkerDto has two instances of WorkerDto and a relationship type field of String.

```
public class WorkerDto {
private Integer id;
private String name;

    // getters and setters

                                        }

public class WorkerWorkerDto {
private WorkerDto workerId1;
private WorkerDto workerId2;
private String relationshipType;

// getters and setters

                                        }
```

Develop the mock service

We will not define the management of the Worker object because we have covered it numerous times in the prior chapters. We will only look at how to accomplish the link between two instances of the Worker and the relationship type. The mock service will be integrated with an in-memory custom database

for persisting the data which will be consumed by the mock user interface. We will initiate with add functionality. We have a list of WorkerWorkerDto, and with every new entry, one instance of WorkerWorkerDto having two instances of WorkerDto and the relationship type is added to the list. A Singleton design pattern with the use of enum is computed in the in-memory database.

```
public enum WorkerWorkerInMemoryDB {

INSTANCE;


private static List<WorkerWorkerDto> list = new
ArrayList<WorkerWorkerDto>();
public void add(WorkerWorkerDto workerWorkerDto) {
list.add(workerWorkerDto);

                                        }

                                        }
```

Now let's review the mock service implementation. The create method adds an instance of WorkerWorkerDto from the user interface and increases the list using the mock database.

```
@Controller
@RequestMapping(value="manytomanyselfreferencewithjoinattributeworkerwor
public class WorkerWorkerMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody WorkerWorkerDto workerWorkerDto){
WorkerWorkerInMemoryDB.INSTANCE.add(workerWorkerDto); }

                                        }
```

Next we will advance to the find all rows service.

```
public enum WorkerWorkerInMemoryDB {
public List<WorkerWorkerDto> findAll() {
return list;

                                        }

                                        }
```

}

In the mock service, the find all rows yields a list of WorkerWorkerDto.

```
public class WorkerWorkerMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public
@ResponseBody List<WorkerWorkerDto> findAll(){
return WorkerWorkerInMemoryDB.INSTANCE.findAll(); }
```

}

Next, we will notice the remove operation which eliminates a specific row with a programmed identifier.

```
public enum WorkerWorkerInMemoryDB {
public void remove(WorkerWorkerDto workerWorkerDto) {
WorkerWorkerDto toRemove = null;
for (WorkerWorkerDto dto:list) {
if (dto.getWorkerId1().getId()==workerWorkerDto.getWorkerId1().getId() &&
dto.getWorkerId2().getId() == workerWorkerDto.getWorkerId2().getId()) {
toRemove = dto;
```

}

}

```
if (toRemove!=null) list.remove(toRemove);
```

}

}

In the mock service, the remove operation excludes a specific row with the distinct identifier provided as an attachment to the url.

```
public class WorkerWorkerMockController {
@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody WorkerWorkerDto workerWorkerDto){
WorkerWorkerInMemoryDB.INSTANCE.remove(workerWorkerDto); }
```

}

Last of all, we generate the isPresent operation accountable for inquiry if a

matching row exists.
public enum WorkerWorkerInMemoryDB {
public boolean isPresent(WorkerWorkerDto workerWorkerDto) {
for (WorkerWorkerDto dto:list) {
if (dto.getWorkerId1().getId() == workerWorkerDto.getWorkerId1().getId() &&
dto.getWorkerId2().getId() == workerWorkerDto.getWorkerId2().getId()) {
return true;
} else if (dto.getWorkerId1().getId() ==
workerWorkerDto.getWorkerId2().getId() && dto.getWorkerId2().getId() ==
workerWorkerDto.getWorkerId1().getId()) {
return true;

}

}

return false;

}

}

The isPresent operation accepts a copy of the WorkerWorkerDto and checks whether a copy of the join prevails.
public class WorkerWorkerMockController {
@RequestMapping(value="/isPresent", method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody WorkerWorkerDto
workerWorkerDto){
return WorkerWorkerInMemoryDB.INSTANCE.isPresent(workerWorkerDto);
}

}

Develop the mock user interface

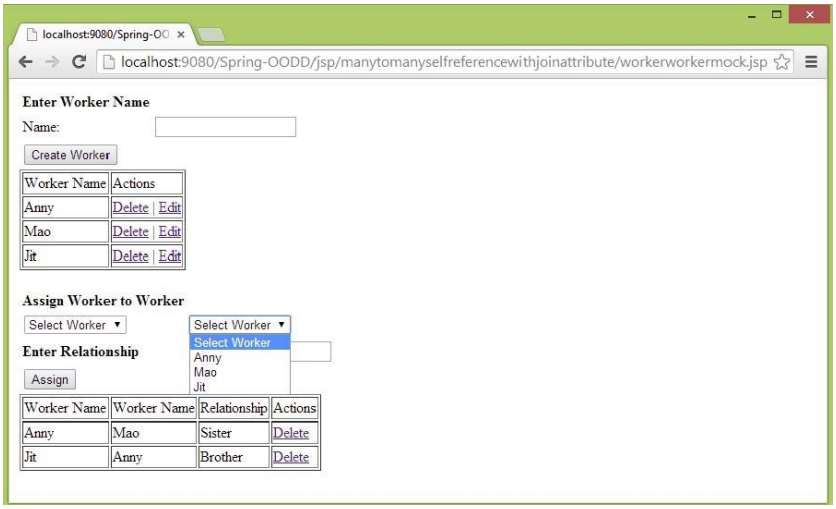

Figure 13-2: A screenshot of the Worker manager user interface In the prevous section we looked at how to cultivate the mock service with an in-memory database. Now we show how to form the user interface shown in Figure 13-2 which will consume this mock service. We will be discussing the join related user interface fragment because the create, read, update and delete methods for the Worker are identical.

The JSP page is the fragment which is executed first and after that the Javascript events are called.

```
<body onload=”loadObjects()”>
<div id=”container”>
<table>
<tbody>
<tr>
<td>
<table>
<tr><td><p><b>Assign Worker to Worker</b></p></td></tr> <tr>
<td>
<div id=”workerCombo1”></div> </td>
<td>
<div id=”workerCombo2”></div> </td>
</tr>
<tr>
<td><p><b>Enter Relationship</b></p></td> <td>
<div id=”relationshipText”></div> </td>
</tr>
<tr>
```

<td><input type=submit value=”Assign” id=”subButtonWorkerWorker” onclick=”return methodCall()”></td> </tr>
</table>
 </td>
</tr>
</tbody>
</table>

</body>
Initially, we will complete the JavaScript loadObjects method which is empowered when the page is set with the onload event.
function loadObjects(){
$.ajax({
url : “*Spring-OODD*manytomanyselfreferencewithjoinattribute/worker/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

processResponseData(data);
processWorker2ResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“workername”).value=””; alert(“Error Status Load Objects:”+textStatus); }

});

$.ajax({
url : “*Spring-OODD*manytomanyselfreferencewithjoinattribute/workerworker/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

```
processWorkerWorkerResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
alert("Error Status Load Objects:"+textStatus); }

});

document.getElementById("relationshipText").innerHTML="<input type='text'
id='relationshipTextBox'/>"; return false;

}
```

The processWorkerWorkerResponseData JavaScript method is called when the above service call is operative. Also the generateWorkerWorkerTableData Javascript method is called which processes the table grid.

```
function processWorkerWorkerResponseData(responsedata){
var dyanamicTableRow="<table border=1>"+
"<tr>" +
"<td>Worker Name</td>"+"<td>Worker
Name</td>"+"<td>Relationship</td>"+"<td>Actions</td>"+
"</tr>";


var dataRow="";
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateWorkerWorkerTableData(itemvalue); });
dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("workerWorkerFormResponse").innerHTML=dyanar
}
function generateWorkerWorkerTableData(itemvalue){
var dataRow="<tr>" +
"<td>" +itemvalue.workerId1.name+"</td>"+
"<td>" +itemvalue.workerId2.name+"</td>"+
"<td>" +itemvalue.relationshipType+"</td>"+
"<td>" +
"<a href=#
onclick=deleteWorkerWorker("+itemvalue.workerId1.id+","+itemvalue.workerI
"</td>"+
"</tr>"
```

¨</tr>´´;
return dataRow;

}

We have mastered the page loading method. If you look closely at how the JSP fragment works, you will realize Javascript methodCall method is being called which calls the create and update for Worker as well as link object.

```
function methodCall(){
var buttonValue = document.getElementById("subButtonWorker").value;
if(buttonValue=="Create Worker"){
createWorker();
} else if(buttonValue=="Update Worker"){
updateWorker();
}
var workerWorkerButtonValue =
document.getElementById("subButtonWorkerWorker").value;
if(workerWorkerButtonValue=="Assign") {
isPresent();

}

return false;

}
```

Next we will launch the isPresent Javascript method which checks whether the join between two instances of Worker already occurs. If the linking does not appear, a new join is formed.

```
function isPresent(){
var workerid1 = $("#workerSelectBox1").val();
var workerid2 = $("#workerSelectBox2").val();
var relationshipType = $("#relationshipTextBox").val();
var workername1 = $("#workerSelectBox1").find('option:selected').text(); var
workername2 = $("#workerSelectBox2").find('option:selected').text();
if(null != workerid1 && "" != workerid1 && null!= workerid2 && "" !=
workerid2 && null!= relationshipType && "" != relationshipType) {
var formData={"workerId1":{"id":workerid1},"workerId2":{"id":workerid2}};
$.ajax({
url : "Spring-
```

*OODD*manytomanyselfreferencewithjoinattribute/workerworker/mock/isPresent’
type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

if(!data) {
createWorkerWorker(workerid1,
workerid2,relationshipType,workername1,workername2); } else {
alert(“Worker already assigned to this Worker”); }
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Create:”+errorThrown);

}

});

}

return false;

}

We will also look at the createWorkerWorker Javascript method which creates the join between the selected two instances of Worker along with the relationship type.
function createWorkerWorker(workerid1,workerid2,relationshipType,
workername1,workername2){
var formData={“workerId1”:{“id”:workerid1,
“name”:workername1},”workerId2”:{“id”:workerid2,
“name”:workername2},”relationshipType”:relationshipType}; if(workerid1 !=
workerid2) {
$.ajax({
url : “*Spring-OODD*manytomanyselfreferencewithjoinattribute/workerworker/mock/create”,

type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert(“Error Status Create:”+textStatus);

}

});

} else {
alert(“Can’t assign worker to itself. Please choose a different worker.”); }

return false;

}

Lastly, we will examine the deleteWorkerWorker Javascript method which takes the two Worker identifiers as a response and ends the relation of the connection.

```
function deleteWorkerWorker(workerid1,workerid2){
var formData={“workerId1”:{“id”:workerid1},”workerId2”:{“id”:workerid2}};
delurl=”Spring-OODDmanytomanyselfreferencewithjoinattribute/workerworker/mock/remove”;
$.ajax({
url : delurl,
type: “POST”,
data : JSON.stringify(formData),
dataType: “json”,
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
```

```
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)
                                                {
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert("Error Status Delete:"+textStatus);
                                                }
                                                });
                                                }
```

Develop the Service

The consequent step is the actual service development which includes four key steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST competencies. We will understand the entity and go up to the REST service.

Develop the resource (entity) tier First, we quantify four instances of Worker and then create two instances of join. Next, we remove one existence and examine whether the join count is singular.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class WorkerWorkerTest {

@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings("unchecked")
@Test
```

```
public void testCRUD() {
Worker worker1 = new Worker();
worker1.setName(“Lalit Narayan Mishra”);
sessionFactory.getCurrentSession().save(worker1);
Worker worker2 = new Worker();
worker2.setName(“Amritendu De”);
sessionFactory.getCurrentSession().save(worker2);
WorkerWorkerId workerWorkerId1 = new WorkerWorkerId(); WorkerWorker
workerWorker1 = new WorkerWorker();
workerWorkerId1.setWorker1(worker1);
workerWorkerId1.setWorker2(worker2);
workerWorker1.setWorkerWorkerId(workerWorkerId1);
workerWorker1.setRelationshipType(“Createspace”);
sessionFactory.getCurrentSession().save(workerWorker1);
Worker worker3 = new Worker(); worker3.setName(“Amish Tripathi”);
sessionFactory.getCurrentSession().save(worker3);
Worker worker4 = new Worker(); worker4.setName(“Amritendu De”);
sessionFactory.getCurrentSession().save(worker4);
WorkerWorkerId workerWorkerId2 = new WorkerWorkerId(); WorkerWorker
workerWorker2 = new WorkerWorker();
workerWorkerId2.setWorker1(worker3);
workerWorkerId2.setWorker2(worker4);
workerWorker2.setWorkerWorkerId(workerWorkerId2);
workerWorker2.setRelationshipType(“Brother”);
sessionFactory.getCurrentSession().save(workerWorker2);
worker3.setName(“Hazekul Alam”);
sessionFactory.getCurrentSession().save(worker3);
List<WorkerWorker> list =
sessionFactory.getCurrentSession().createQuery(“from WorkerWorker”).list();
Assert.assertEquals(2L, list.size());
List<WorkerWorker> workerWorkerList =
sessionFactory.getCurrentSession().createQuery(“from WorkerWorker”).list();
WorkerWorker tmpWorkerWorker = workerWorkerList.get(0);
sessionFactory.getCurrentSession().delete(tmpWorkerWorker);
List<WorkerWorker> list2 =
sessionFactory.getCurrentSession().createQuery(“from WorkerWorker”).list();
Assert.assertEquals(1L, list2.size()); }
}
```

We will review the Worker entity which comprises an identifier and a name String field.

```
@Entity
@Table(name=”WORKER”)
public class Worker {


private Integer id;
private String name;
private Set<WorkerWorker> workers1 = new HashSet<WorkerWorker>(0);
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name=”ID”)
public Integer getId() {
return id;

                                        }

public void setId(Integer id) {
this.id = id;

                                        }

@Column(name=”NAME”, nullable=false, length=100) public String
getName() {
return name;

                                        }

public void setName(String name) {
this.name = name;

                                        }

@OneToMany(fetch = FetchType.LAZY, mappedBy =
“workerWorkerId.worker1”, cascade=CascadeType.ALL) public
Set<WorkerWorker> getWorkers1() {
return workers1;

                                        }
```

```
public void setWorkers1(Set<WorkerWorker> workers1) {
this.workers1 = workers1;

                                                    }

                                                    }
```

Subsequently, we will review the Worker Worker entity which comprises two instances of Worker provided by WorkerWorkerId and the relationship type.

```
@Entity
@Table(name=”WORKER_WORKER”)
@AssociationOverrides({
   @AssociationOverride(name = “workerWorkerId.worker1”, joinColumns =
@JoinColumn(name = “WORKER1_ID”, referencedColumnName=”ID”)),
@AssociationOverride(name = “workerWorkerId.worker2”, joinColumns =
@JoinColumn(name = “WORKER2_ID”, referencedColumnName=”ID”)) })
public class WorkerWorker implements Serializable {


private static final long serialVersionUID = 2165276994610614854L;
private WorkerWorkerId workerWorkerId = new WorkerWorkerId(); private
String relationshipType;

@EmbeddedId
public WorkerWorkerId getWorkerWorkerId() {
return workerWorkerId;

                                                    }

public void setWorkerWorkerId(WorkerWorkerId workerWorkerId) {
this.workerWorkerId = workerWorkerId;

                                                    }

@Transient
public Worker getWorker1() {
return getWorkerWorkerId().getWorker1();

                                                    }
```

```
public void setWorker1(Worker worker1) {
getWorkerWorkerId().setWorker1(worker1);

                                                    }

@Transient
public Worker getWorker2() {
return getWorkerWorkerId().getWorker2();

                                                    }


public void setWorker2(Worker worker2) {
getWorkerWorkerId().setWorker1(worker2);

                                                    }

@Column(name=”RELATIONSHIP_TYPE”, nullable=false, length=100)
public String getRelationshipType() {
return relationshipType;

                                                    }

public void setRelationshipType(String relationshipType) {
this.relationshipType = relationshipType;

                                                    }

                                                    }
```

Finally, we will review the @Embeddable WorkerWorkerId class which comprises two @ManyToOne instances of Worker.

```
@Embeddable
public class WorkerWorkerId implements Serializable {


private static final long serialVersionUID = -29551278665987254l4L;
private Worker worker1;
private Worker worker2;
```

```
private Worker worker2;

@ManyToOne
public Worker getWorker1() {
return worker1;

                                        }

public void setWorker1(Worker worker1) {
this.worker1 = worker1;

                                        }

@ManyToOne
public Worker getWorker2() {
return worker2;

                                        }


public void setWorker2(Worker worker2) {
this.worker2 = worker2;

                                        }

                                        }
```

Table structure

```
DROP TABLE `worker_worker`;
DROP TABLE `worker`;
CREATE TABLE `worker` (
  `ID` int(11) NOT NULL AUTO_INCREMENT, `NAME` varchar(100) NOT
NULL,
  PRIMARY KEY (`ID`)

                                        );

CREATE TABLE `worker_worker` (
  `RELATIONSHIP_TYPE` varchar(100) NOT NULL, `WORKER2_ID` int(11)
```

NOT NULL DEFAULT ‘0’, `WORKER1_ID` int(11) NOT NULL DEFAULT ‘0’, PRIMARY KEY (`WORKER1_ID`,`WORKER2_ID`), KEY (`WORKER2_ID`),
KEY (`WORKER1_ID`),
CONSTRAINT FOREIGN KEY (`WORKER1_ID`) REFERENCES `worker` (`ID`), CONSTRAINT FOREIGN KEY (`WORKER2_ID`) REFERENCES `worker` (`ID`));

Develop the data access tier

Next we will get into how to grow the data access tier. We will jump to the test case for the get all records package. Please note that we will skip over the Worker link data access tier. The Worker data access tier remains unaffected as covered in prior chapters.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class WorkerWorkerDaoImplTest {

@Autowired
private WorkerDao workerDao;

@Autowired
private WorkerWorkerDao workerWorkerDao;

@Test
public void testGetAll() {
Assert.assertEquals(0L, workerWorkerDao.getAll().size()); }

}
```

We will refer to how to cultivate the corresponding data access operation for the test revealed above. The Hibernate Session.createQuery method is used with an SQL to retrieve all the records from the WorkerWorker entity.

```
@Repository
@Transactional
public class WorkerWorkerDaoImpl implements WorkerWorkerDao {

@Autowired
```

```
private SessionFactory sessionFactory ;

@SuppressWarnings("unchecked")
@Override
public List<WorkerWorker> getAll() {
return sessionFactory.getCurrentSession().createQuery("select distinct ww from
WorkerWorker ww").list(); }

                                                    }
```

Let's now see the test for isPresent method. We insert two instances of Worker and link them. After that we check if the isPresent resort is true.

```
public class WorkerWorkerDaoImplTest {

@Autowired
private WorkerDao workerDao;

@Autowired
private WorkerWorkerDao workerWorkerDao;

@Test
public void testIsPresent() {
boolean status = false;
Worker worker1 = new Worker();
worker1.setName("Lalit Narayan Mishra");
workerDao.insert(worker1);


Worker worker2 = new Worker();
worker2.setName("Amritendu De");
workerDao.insert(worker2);


WorkerWorkerId workerWorkerId = new WorkerWorkerId(); WorkerWorker
workerWorker = new WorkerWorker(); workerWorkerId.setWorker1(worker1);
workerWorkerId.setWorker2(worker2);
workerWorker.setWorkerWorkerId(workerWorkerId);
workerWorker.setRelationshipType("Colleagues");
workerWorkerDao.insert(workerWorker);
```

workerWorkerDao.insert(workerWorker);

Worker worker3 = workerDao.getAll().get(0);
Worker worker4 = workerDao.getAll().get(1);

List<WorkerWorker> workerWorkerList =
workerWorkerDao.isPresent(worker4.getId(), worker3.getId()); if(null !=
workerWorkerList) {
if(workerWorkerList.size() > 0) {
status = true;

}

}

Assert.assertTrue(status);

}

}

The mirroring isPresent operation just requests the Hibernate Session.createQuery method, passing the instance identifiers, and tests whether a join exists.

```
public class WorkerWorkerDaoImpl implements WorkerWorkerDao {

@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings("unchecked")
@Override
public List<WorkerWorker> isPresent(Integer workerId1, Integer workerId2) {
String hql = "select distinct ww from WorkerWorker ww where
ww.workerWorkerId.worker1.id=:workerId1 and
ww.workerWorkerId.worker2.id=:workerId2"; Query query =
sessionFactory.getCurrentSession().createQuery(hql);
query.setParameter("workerId1", workerId1);
query.setParameter("workerId2", workerId2);
```

```
return query.list();

                                                    }

                                                    }
```

Develop the business service tier

We will move up to the business service tier. First, we will review the test for fetching all records job.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class WorkerWorkerServiceImplTest {

@Autowired
private WorkerService workerService;

@Autowired
private WorkerWorkerService workerWorkerService;
@Test
public void testFindAll() {
Assert.assertEquals(0L, workerWorkerService.findAll().size()); }

                                                    }
```

Let us examine how to program the related method in the business service.

```
@Service
@Transactional
public class WorkerWorkerServiceImpl implements WorkerWorkerService {

@Autowired
private WorkerDao workerDao;

@Autowired
private WorkerMapper workerMapper;

@Autowired
private WorkerWorkerDao workerWorkerDao;
```

```
@Override
public List<WorkerWorkerDto> findAll() {
List<WorkerWorkerDto> workerWorkerDtos = new
ArrayList<WorkerWorkerDto>(); List<WorkerWorker> workerList =
workerWorkerDao.getAll(); for(WorkerWorker workerWorker : workerList) {
WorkerWorkerDto workerWorkerDto = new WorkerWorkerDto();
workerWorkerDto.setWorkerId1(workerMapper.mapEntityToDto(workerWorke
workerWorkerDto.setWorkerId2(workerMapper.mapEntityToDto(workerWorke
workerWorkerDto.setRelationshipType(workerWorker.getRelationshipType());
workerWorkerDtos.add(workerWorkerDto);

                                                    }

return workerWorkerDtos;

                                                    }

                                                    }
```

Moving on to the create test operation which creates one occurrence of the Worker link object and checks if the number of relationships is limited with the fetch all records operation.

```
public class WorkerWorkerServiceImplTest {

@Autowired
private WorkerService workerService;

@Autowired
private WorkerWorkerService workerWorkerService;
@Test
public void testCreate() {
WorkerWorkerDto workerWorkerDto = new WorkerWorkerDto();
WorkerDto workerDto1 = new WorkerDto();
workerDto1.setName(“The Immortals of Meluha”);
workerService.create(workerDto1);



WorkerDto workerDto2 = new WorkerDto();
workerDto2.setName(“Amish Tripathi”);
```

workerDto2.setName("Anish Tripathi");
workerService.create(workerDto2);

List<WorkerDto> workerDtos1 = workerService.findAll(); WorkerDto workerDto3 = workerDtos1.get(0);

List<WorkerDto> workerDtos2 = workerService.findAll(); WorkerDto workerDto4 = workerDtos2.get(0);

workerWorkerDto.setWorkerId1(workerDto3);
workerWorkerDto.setWorkerId2(workerDto4);
workerWorkerDto.setRelationshipType("Friends");
workerWorkerService.create(workerWorkerDto);
Assert.assertEquals(1L, workerWorkerService.findAll().size()); }

}

The corresponding method in the business service tier keeps a Worker linked data access object.
public class WorkerWorkerServiceImpl implements WorkerWorkerService {

@Autowired
private WorkerDao workerDao;

@Autowired
private WorkerMapper workerMapper;

@Autowired
private WorkerWorkerDao workerWorkerDao;

@Override
public void create(WorkerWorkerDto workerWorkerDto) {
Integer workerId1 = workerWorkerDto.getWorkerId1().getId(); Integer workerId2 = workerWorkerDto.getWorkerId2().getId();
Worker worker1 = workerDao.getById(workerId1); Worker worker2 = workerDao.getById(workerId2);

WorkerWorkerId workerWorkerId = new WorkerWorkerId(); WorkerWorker workerWorker = new WorkerWorker(); workerWorkerId.setWorker1(worker1); workerWorkerId.setWorker2(worker2);
workerWorker.setWorkerWorkerId(workerWorkerId);
workerWorker.setRelationshipType(workerWorkerDto.getRelationshipType());
workerWorkerDao.insert(workerWorker);

}

}

The remove test method is next in line, which creates a link between two instances of Worker followed by a link delete and then checks whether there is no link between the two items.

```
public class WorkerWorkerServiceImplTest {

@Autowired
private WorkerService workerService;

@Autowired
private WorkerWorkerService workerWorkerService;
@Test
public void testRemove() {
WorkerWorkerDto workerWorkerDto = new WorkerWorkerDto();
WorkerDto workerDto1 = new WorkerDto();
workerDto1.setName("The Immortals of Meluha");
workerService.create(workerDto1);


WorkerDto workerDto2 = new WorkerDto();
workerDto2.setName("Amish Tripathi");
workerService.create(workerDto2);


List<WorkerDto> workerDtos1 = workerService.findAll(); WorkerDto
workerDto3 = workerDtos1.get(0);


List<WorkerDto> workerDtos2 = workerService.findAll(); WorkerDto
```

List<WorkerDto> workerDtos2 = workerService.findAll(); WorkerDto workerDto4 = workerDtos2.get(0);

workerWorkerDto.setWorkerId1(workerDto3);
workerWorkerDto.setWorkerId2(workerDto4);
workerWorkerDto.setRelationshipType("Friends");
workerWorkerService.create(workerWorkerDto);
Assert.assertEquals(1L, workerWorkerService.findAll().size());
List<WorkerWorkerDto> workerWorkerList = workerWorkerService.findAll();
WorkerWorkerDto workerWorkerDto1 = workerWorkerList.get(0);
workerWorkerService.remove(workerWorkerDto1);

Assert.assertEquals(0, workerWorkerService.findAll().size()); }

}

The associated business service method takes a linked Worker object and removes the link from the list by means of the data access method.

```
public class WorkerWorkerServiceImpl implements WorkerWorkerService {

@Autowired
private WorkerDao workerDao;

@Autowired
private WorkerMapper workerMapper;

@Autowired
private WorkerWorkerDao workerWorkerDao;

@Override
public void remove(WorkerWorkerDto workerWorkerDto) {
Integer workerId1 = workerWorkerDto.getWorkerId1().getId(); Integer
workerId2 = workerWorkerDto.getWorkerId2().getId();
Worker worker1 = workerDao.getById(workerId1); Worker worker2 =
workerDao.getById(workerId2);
List<WorkerWorker> workerWorkerList =
workerWorkerDao.isPresent(workerId1, workerId2); for(WorkerWorker
```

```
workerWorker : workerWorkerList) {
worker1.getWorkers1().remove(workerWorker);
worker2.getWorkers1().remove(workerWorker);
workerWorkerDao.delete(workerWorker);

                                                    }

workerDao.update(worker1);
workerDao.update(worker2);

                                                    }

}
```

The finishing test operation is the isPresent action which creates a linked Worker and then reviews whether there is any link between the two objects.

```
public class WorkerWorkerServiceImplTest {

@Autowired
private WorkerService workerService;

@Autowired
private WorkerWorkerService workerWorkerService;
@Test
public void testIsPresent() {
WorkerWorkerDto workerWorkerDto = new WorkerWorkerDto();
WorkerDto workerDto1 = new WorkerDto();
workerDto1.setName(“The Immortals of Meluha”);
workerService.create(workerDto1);



WorkerDto workerDto2 = new WorkerDto();
workerDto2.setName(“Amish Tripathi”);
workerService.create(workerDto2);



List<WorkerDto> workerDtos1 = workerService.findAll(); WorkerDto
workerDto3 = workerDtos1.get(0);
```

```
List<WorkerDto> workerDtos2 = workerService.findAll(); WorkerDto
workerDto4 = workerDtos2.get(0);


workerWorkerDto.setWorkerId1(workerDto3);
workerWorkerDto.setWorkerId2(workerDto4);
workerWorkerDto.setRelationshipType("Friends");
workerWorkerService.create(workerWorkerDto);
Assert.assertEquals(1L, workerWorkerService.findAll().size());
boolean status = workerWorkerService.isPresent(workerWorkerDto);
Assert.assertTrue(status);

                                                    }

                                                    }
```

The comparable operation of the business service is the isPresent job which delegates the call to the data access, the isPresent method.

```
public class WorkerWorkerServiceImpl implements WorkerWorkerService {

@Autowired
private WorkerDao workerDao;

@Autowired
private WorkerMapper workerMapper;

@Autowired
private WorkerWorkerDao workerWorkerDao;



@Override
public boolean isPresent(WorkerWorkerDto workerWorkerDto) {
boolean status = false;
List<WorkerWorker> workerWorkerList =
workerWorkerDao.isPresent(workerWorkerDto.getWorkerId1().getId(),
workerWorkerDto.getWorkerId2().getId()); if(workerWorkerList.size() > 0) {
status = true;
} else {
```

```
} else {
workerWorkerList =
workerWorkerDao.isPresent(workerWorkerDto.getWorkerId2().getId(),
workerWorkerDto.getWorkerId1().getId()); if(workerWorkerList.size() > 0) {
status = true;

                                                    }

                                                    }

return status;

                                                    }

                                                    }
```

Develop the presentation tier

The REST based Spring Controller is next in the queue for which we will first do the test. The test develops a create link object, and then checks if it is present followed by a delete action.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class WorkerWorkerControllerTest {
private Gson gson = new GsonBuilder().setDateFormat(“yyyy-MM-dd
hh:mm:ss”).create();
@Autowired
private WorkerService workerService;

@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc = MockMvcBuilders.
<StandaloneMockMvcBuilder>webAppContextSetup
(webApplicationContext).build();
```

}

@Test
public void testAll() throws Exception {
testCreate();
testPresent();
testDelete();

}

public void testCreate() throws Exception {
WorkerWorkerDto workerWorkerDto = new WorkerWorkerDto();
WorkerDto workerDto1 = new WorkerDto();
workerDto1.setName(“Amritendu De”);
workerService.create(workerDto1);

WorkerDto workerDto2 = new WorkerDto();
workerDto2.setName(“Amish Tripathi”);
workerService.create(workerDto2);

List<WorkerDto> workerDtos1 = workerService.findAll(); WorkerDto
workerDto3 = workerDtos1.get(0);

List<WorkerDto> workerDtos2 = workerService.findAll(); WorkerDto
workerDto4 = workerDtos2.get(0);

workerWorkerDto.setWorkerId1(workerDto3);
workerWorkerDto.setWorkerId2(workerDto4);
workerWorkerDto.setRelationshipType(“Authors”);
String json = gson.toJson(workerWorkerDto);

MockHttpServletRequestBuilder requestBuilderOne =

MockMvcRequestBuilders.post("*manytomanyselfreferencewithjoinattribute*work
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}

public void testPresent() throws Exception {
WorkerWorkerDto workerWorkerDto = new WorkerWorkerDto();
List<WorkerDto> workerDtos1 = workerService.findAll(); WorkerDto
workerDto3 = workerDtos1.get(0);

List<WorkerDto> workerDtos2 = workerService.findAll(); WorkerDto
workerDto4 = workerDtos2.get(0);

workerWorkerDto.setWorkerId1(workerDto3);
workerWorkerDto.setWorkerId2(workerDto4);
workerWorkerDto.setRelationshipType("Authors");
String json = gson.toJson(workerWorkerDto);

MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post("*manytomanyselfreferencewithjoinattribute*work
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}

public void testDelete() throws Exception {
WorkerWorkerDto workerWorkerDto = new WorkerWorkerDto();
List<WorkerDto> workerDtos1 = workerService.findAll(); WorkerDto
workerDto3 = workerDtos1.get(0);

List<WorkerDto> workerDtos2 = workerService.findAll(); WorkerDto workerDto4 = workerDtos2.get(0);

workerWorkerDto.setWorkerId1(workerDto3);
workerWorkerDto.setWorkerId2(workerDto4);
workerWorkerDto.setRelationshipType("Authors");
String json = gson.toJson(workerWorkerDto);

MockHttpServletRequestBuilder requestBuilder2 = MockMvcRequestBuilders.post("*manytomanyselfreferencewithjoinattribute*work
requestBuilder2.contentType(MediaType.APPLICATION_JSON);
requestBuilder2.content(json.getBytes());
this.mockMvc.perform(requestBuilder2).andExpect(MockMvcResultMatchers.st
}

}

The REST, Spring Controller will be networking with the business service tier which delegates the call to the data access tier tied to the database. The setup is like the mock service, with the change that the code is working with the genuine database and not the in memory catalog.

```
@Controller
@RequestMapping(value="manytomanyselfreferencewithjoinattributeworkerwor
@Transactional
public class WorkerWorkerController {

@Autowired
private WorkerWorkerService service;

@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody List<WorkerWorkerDto> findAll(){
return service.findAll();

}
```

```
@RequestMapping(value=”/isPresent”, method=RequestMethod.POST) public
@ResponseBody boolean isPresent(@RequestBody WorkerWorkerDto
workerWorkerDto){
return service.isPresent(workerWorkerDto);

}

@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody WorkerWorkerDto workerWorkerDto){
service.create(workerWorkerDto);

}

@RequestMapping(value=”/remove”, method=RequestMethod.POST)
@ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@RequestBody WorkerWorkerDto workerWorkerDto){
service.remove(workerWorkerDto);

}

}
```

The pivotal step in the development is the collaborating of the REST service with the actual user interface. Please note that the request mapping changes from ”*manytomanyselfreferencewithjoinattribute*workerworker/mock” to “*manytomanyselfreferencewithjoinattribute*workerworker”. The outstanding task involves forming the actual user interface using the mock user interface.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to a many-to-many selfreferencing with join attribute
- Develop data access tier

- Develop business service tier
- Develop a REST service controller.

Part V. Managing Inheritance Relationships

Chapter 14. Single Table Inheritance

This chapter is a primer to the process of development with a single table inheritance relationship. We will continue our journey of creating the user interface at the beginning and then write the real service. At the end, we will be integrating both the solutions and arriving at a reasonable conclusion.

Domain Model

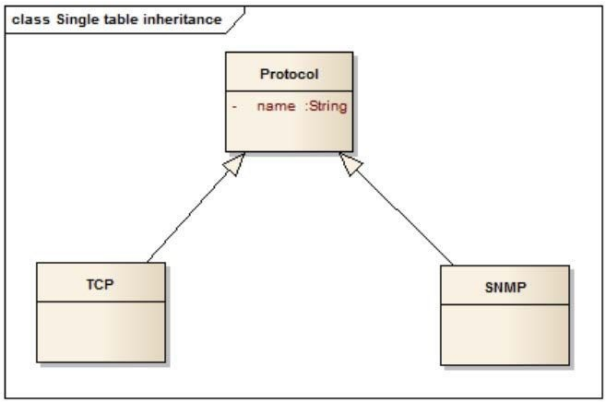

Figure 14-1: A Single Table Inheritance relationship Different types of objects may have something in common with each other. Object orientation allows classes to inherit common attributes and methods which is referred to as Inheritance. Generalization is the art of extracting shared attributes and methods from a related set of classes and forming a generalized superclass. On the other hand, Specialization means forming a subclass with attributes and methods specific to the subclass. In this chapter, we will look at the first approach to

handling inheritance in object relational modeling. We will begin with a single table inheritance relationship, as shown in Figure 14-1. The Protocol entity has an identifier, a name, and a field known as discriminator which denotes the difference between an SNMP and a TCP. Here the generalized superclass is Protocol and the specialized subclasses are SNMP and TCP. Let's start by discussing how the user interface may be developed.

Develop a User Interface

We will divide the user interface development into three main tasks – the data transfer object, the mock service and the mock user interface. We will look at each of these tasks one by one.

Develop the data transfer object

The data transfer object contains ProtocolDto which has an identifier and a name. There are two sub-types of ProtocolDto known as TCPDto and SNMPDto. The @JsonTypeInfo annotation is used to denote Polymorphic types and the kind of metadata used for serializing and deserializing type information. The @JsonSubTypes annotation is used to denote sub-types of serializable polymorphic types.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include =
JsonTypeInfo.As.PROPERTY, property = "type") @JsonSubTypes({
   @Type(value = TCPDto.class, name = "tcp"), @Type(value =
SNMPDto.class, name = "snmp") }) public class ProtocolDto {
private Integer id;
private String name;

  // getters and setters
 }
public class TCPDto extends ProtocolDto {

                                                    }

public class SNMPDto extends ProtocolDto {

                                                    }
```

}

Develop the mock service

Let’s start with a simple add to the in-memory custom database which is based on the Singleton pattern implemented as an enum.

```
public enum ProtocolInMemoryDB {

INSTANCE;


private static List<ProtocolDto> list = new ArrayList<ProtocolDto>(); private static Integer lastId = 0;


public Integer getId() {
return ++lastId;

}

public void add(ProtocolDto protocolDto) {
protocolDto.setId(getId());
list.add(protocolDto);

}

}
```

We will be looking at the mock service for create functionality which uses the above in-memory database for persisting the data.

```
@Controller
@RequestMapping(value="singletableinheritancemock") public class
ProtocolMockController {

@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ProtocolDto protocol){
ProtocolInMemoryDB.INSTANCE.add(protocol);

}
```

}

}

Next we will look at the method which is responsible for retrieval of all the rows. First, we will look at the in-memory database operations which the controller is going to use.

```
public enum ProtocolInMemoryDB {


public List<ProtocolDto> findAll() {
return list;

}

public ProtocolDto findById(Integer id) {
for (ProtocolDto dto:list) {
if (dto.getId()==id)
return dto;

}

return null;

}

}
```

The *findAll returns all instances of sub-types of ProtocolDto and* findById returns a specific row with the given identifier.

```
public class ProtocolMockController {


@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody ProtocolDto[] findAll(){
List<ProtocolDto> list = ProtocolInMemoryDB.INSTANCE.findAll(); return
list.toArray(new ProtocolDto[list.size()]); }
@RequestMapping(value="/findById/{protocolid}",
method=RequestMethod.GET) public @ResponseBody ProtocolDto
findById(@PathVariable("protocolid") Integer protocolid){
```

```
return ProtocolInMemoryDB.INSTANCE.findById(protocolid); }

                                                    }
```

Next we will look at the remove operation which removes a specific row with an identifier.

```
public enum ProtocolInMemoryDB {

public void remove(Integer id) {
ProtocolDto toRemove = null;
for (ProtocolDto dto:list)
if (dto.getId()==id) toRemove = dto;
if (toRemove!=null) list.remove(toRemove);

                                                    }

                                                    }
```

The /remove operation deletes a specific row with the given identifier.

```
public class ProtocolMockController {
@RequestMapping(value=”/remove/{protocolid}”,
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“protocolid”) Integer protocolid){
ProtocolInMemoryDB.INSTANCE.remove(protocolid); }

                                                    }
```

The edit method updates a specific row with the details. Let’s look at the in-memory database operations which the controller is going to use.

```
public class ProtocolInMemoryDB {
public void edit(ProtocolDto protocolDto) {
for (ProtocolDto dto:list) {
if (dto.getId()==protocolDto.getId())
dto.setName(protocolDto.getName());

                                                    }

}

                                                    }
```

}

The /edit operation updates a row with details using HTTP POST method.

```
public class ProtocolMockController {
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody ProtocolDto protocol){
ProtocolInMemoryDB.INSTANCE.edit(protocol);
}
```

}

Develop the mock user interface

Let’s look at creating the user interface shown in Figure 14-2.

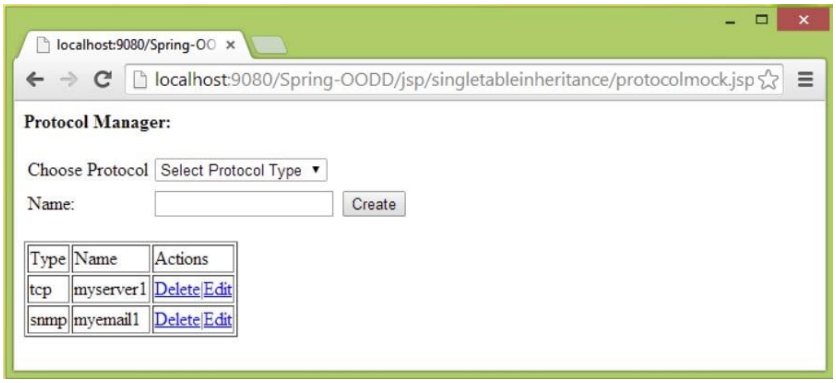

Figure 14-2: A screenshot of the Protocol manager user interface Let’s first look at the JSP fragment which constitutes the body of the page.

```
<body onload=”loadObjects()”>
<div id=”container”>
<div>
<p><b>Protocol Manager:</b></p> </div>
<form method=”post” id=”protocolForm”>
<table>
<tbody>
<tr>
<td>Choose Protocol</td>
<td>
<select name=”protocolChooser” id=”protocolChooser”> <option
```

value=´´´>Select Protocol Type</option> <option value=´tcp´>TCP</option>
<option value=”snmp”>SNMP</option> </select>
</td>
</tr>
<tr>
<td>Name:</td>
<td>
<input name=”protocolid” id=”protocolid” type=”hidden”> <input
name=”protocolname” id=”protocolname” type=”text”> </td>
<td><input type=submit value=”Create” id=”subButton” onclick=”return
methodCall()”></td> </tr>
</tbody>
</table>
</form>

</body>
First we will see the JavaScript method which is called when the page is loaded with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDsingletableinheritance/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)
{
processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“protocolname”).value=””; alert(“Error Status Load
Objects:”+textStatus); }
});
$(‘#protocolChooser’).get(0).selectedIndex = 0; return false;
}
```

The JavaScript processResponseData method is called when the above service

call is successful. Also the Javascript method generateTableData is called to form the table grid after the values are retrieved.

```
function processResponseData(data){
var dyanamicTableRow="<table border=1>"+
"<tr>" +
"<td>Type</td>"+
"<td>Name</td>"+
"<td>Actions</td>"+
"</tr>";
var dataRow="";
$.each(data, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue);

});

dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("protocolFormResponse").innerHTML=dyanamicTal
}
function generateTableData(itemvalue){
var dataRow="<tr>" +
"<td>" +itemvalue.type+"</td>"+
"<td>" +itemvalue.name+"</td>"+
"<td>" +
"<a href=# onclick=deleteObject("+itemvalue.id+")>Delete</a>"+
"|<a href=# onclick=editObject("+itemvalue.id+")>Edit</a>"+
"</td>"+
"</tr>";
return dataRow;

}
```

We have witnessed how the page loading works. Now let's look at the create and update functionality called using the Javascript methodCall method.

```
function methodCall(){
var buttonValue = document.getElementById("subButton").value;
if(buttonValue=="Create"){
create();
}else if(buttonValue=="Update"){
update();
```

}

return false;

}

First, we will look at the Javascript create method.

```
function create(){
var name = $(“#protocolname”).val();
var type = $(“#protocolChooser”).val();


if(type == “” || type == null) {
alert(“Please choose Protocol Type”);
return false;

}

if(name == “” || name == null) {
alert(“Please enter Protocol name”);
return false;

}

var formData={“name”:name,”type”:type};
$.ajax({
url : “Spring-OODDsingletableinheritance/mock/create”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

document.getElementById(“protocolname”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“protocolname”).value=””; alert(“Error Status
```

Create:”+textStatus);

}

});

return false;

}

Next, we will look at the Javascript update method.
function update(){
var name = $(“#protocolname”).val();
var id = +$(“#protocolid”).val();
var type = $(“#protocolChooser”).val();
var formData={“id”:id,”name”:name,”type”:type}; $.ajax({
url : “*Spring-OODD*singletableinheritance/mock/edit”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

{

document.getElementById(“protocolname”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“protocolname”).value=””; alert(“Error Status
Update:”+textStatus);

}

});

return false;

}

The editObject and the viewObject Javascript method call allows the selection of a record for an edit operation.

```
function editObject(protocolid){
var editurl="Spring-OODDsingletableinheritance/mock/findById/"+protocolid;
var protocolForm={id:protocolid};
$.ajax({
url : editurl,
type: "GET",
data : protocolForm,
dataType: "json",
success: function(data, textStatus, jqXHR)

{

viewObject(data);
$("#protocolChooser").val(data.type);
document.getElementById("subButton").value="Update";
},
error: function (jqXHR, textStatus, errorThrown) {
alert("Error Status Find Object:"+textStatus); }

});

}

function viewObject(data){
document.getElementById("protocolname").value=data.name;
document.getElementById("protocolid").value=data.id; }
```

Lastly, we will look at the Javascript deleteObject method.

```
function deleteObject(protocolid){
var protocolForm={id:protocolid};
var delurl="Spring-OODDsingletableinheritance/mock/remove/"+protocolid;
$.ajax({
url : delurl,
type: "POST",
data : protocolForm,
dataType: "json",
success: function(data, textStatus, jqXHR)

{
```

```
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert("Error Status Delete:"+textStatus);

}

});

}
```

Develop the Service

The service development will be distributed into four major steps – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST service which will be integrated with the user interface. Let's go one at a time and let's go bottom up, starting with the entity and going up to the REST service.

Develop the resource (entity) tier First, we will look at the test for Protocol entity which constitutes basic operations like insert, update, delete and find.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ProtocolTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings("unchecked")
@Test
public void testCRUD()

{

Protocol p1 = new Protocol();
```

```
Protocol p1 = new Protocol();
p1.setName("A");


Protocol p2 = new Protocol();
p2.setName("B");


sessionFactory.getCurrentSession().save(p1);
sessionFactory.getCurrentSession().save(p2);


p1.setName("C");
sessionFactory.getCurrentSession().merge(p1);

List<Protocol> list = sessionFactory.getCurrentSession().createQuery("from
Protocol").list(); Assert.assertEquals(2L, list.size());


sessionFactory.getCurrentSession().delete(p1);
List<Protocol> list2 = sessionFactory.getCurrentSession().createQuery("from
Protocol").list(); Assert.assertEquals(1L, list2.size());

                                                    }

                                                    }
```

We will look at the definition of Protocol entity. The @Inheritance annotation defines the inheritance strategy which is a Single table inheritance for this example. The @DiscriminatorColumn annotation defines the name of the column discriminator and the type which is String in this case. The @DiscriminatorValue annotation defines the value of the discriminator column for entities of a particular type. For example, the root class discriminator value is "PROTOCOL" and the two sub-types are "TCP" and "SNMP".

```
@Entity
@Table(name = "PROTOCOL")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "discriminator", discriminatorType =
DiscriminatorType.STRING) @DiscriminatorValue(value = "PROTOCOL")
```

```
public class Protocol {

private Integer id;
private String name;

@Id
@GeneratedValue(strategy = GenerationType.AUTO) @Column(name = “ID”)
public Integer getId() {
return id;

                                        }


public void setId(Integer id) {
this.id = id;

                                        }


@Column(name = “NAME”, nullable = false, length = 100) public String
getName() {
return name;

                                        }


public void setName(String name) {
this.name = name;

                                        }

                                        }
```

Next, we will look at the test for the TCP entity.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class TCPTest {

@Autowired
```

```
private SessionFactory sessionFactory;

@SuppressWarnings("unchecked")
@Test
public void testCRUD()

                                        {

Protocol p1 = new Protocol();
p1.setName("Protocol");
sessionFactory.getCurrentSession().save(p1);



TCP tcp = new TCP();
tcp.setName("TCP");
sessionFactory.getCurrentSession().save(tcp);



tcp.setName("TCP/IP");
sessionFactory.getCurrentSession().merge(tcp);
List<TCP> list = sessionFactory.getCurrentSession().createQuery("from
TCP").list(); Assert.assertEquals(1L, list.size());



sessionFactory.getCurrentSession().delete(tcp);
List<TCP> list2 = sessionFactory.getCurrentSession().createQuery("from
TCP").list(); Assert.assertEquals(0L, list2.size());

                                        }

                                        }
```

We review the definition of the TCP entity based on the test above.

```
@Entity
@Table(name="PROTOCOL")
@DiscriminatorValue("TCP")
public class TCP extends Protocol {

                                        }
```

Lastly, we foresee the test for the SNMP entity which is similar to the test for the TCP entity.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class SNMPTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
public void testCRUD()

                                                {

Protocol p1 = new Protocol();
p1.setName(“Protocol”);
sessionFactory.getCurrentSession().save(p1);



SNMP snmp = new SNMP();
snmp.setName(“SNMP”);
sessionFactory.getCurrentSession().save(snmp);
snmp.setName(“SNMP Protocol”);
sessionFactory.getCurrentSession().merge(snmp);
List<SNMP> list = sessionFactory.getCurrentSession().createQuery(“from
SNMP”).list(); Assert.assertEquals(1L, list.size());



sessionFactory.getCurrentSession().delete(snmp);
List<SNMP> list2 = sessionFactory.getCurrentSession().createQuery(“from
SNMP”).list(); Assert.assertEquals(0L, list2.size());

                                                }

                                                }
```

Finally, we look at the SNMP entity definition which is similar to the TCP entity with the difference in discriminator value.

```
@Entity
@Table(name=”PROTOCOL”)
@DiscriminatorValue(“SNMP”)
public class SNMP extends Protocol {

}
```

Table structure

```
DROP TABLE `protocol`;
CREATE TABLE `protocol` (
  `discriminator` varchar(31) NOT NULL, `ID` int(11) NOT NULL
AUTO_INCREMENT, `NAME` varchar(100) NOT NULL,
  PRIMARY KEY (`ID`)

) ;
```

Develop the data access tier

We will start with the test case for the find all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ProtocolDaoImplTest {
@Autowired
private ProtocolDao dao ;

@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size());

}

}
```

We will see how to write the corresponding data access operation of the above test.

```
@Repository
```

```
@Transactional
public class ProtocolDaoImpl  implements ProtocolDao {
@Autowired
private SessionFactory sessionFactory ;

@SuppressWarnings(“unchecked”)
@Override
public List<Protocol> getAll() {
return sessionFactory.getCurrentSession().createQuery(“select protocol from
Protocol protocol order by protocol.id desc”).list(); }

                                                    }
```

Let’s move on to the insert operation test.

```
public class ProtocolDaoImplTest {
@Autowired
private ProtocolDao dao ;

@Test
public void testInsert() {
Protocol tprotocol = new Protocol();
tprotocol.setName(“TCP/IP”);
dao.insert(tprotocol);



Protocol sProtocol = new Protocol();
sProtocol.setName(“SNMP”);
dao.insert(sProtocol);



Assert.assertEquals(2L, dao.getAll().size());

                                                    }

                                                    }
```

The corresponding insert operation in the data access class has to be implemented using Hibernate 4 Session. save () function.

```
public class ProtocolDaoImpl  implements ProtocolDao {
```

```
@Autowired
private SessionFactory sessionFactory ;
@Override
public void insert(Protocol protocol) {
sessionFactory.getCurrentSession().save(protocol); }

                                                    }
```

We will jump to the next operation which is finding a row with an identifier.

```
public class ProtocolDaoImplTest {
@Test
public void testGetById() {
Protocol sprotocol = new Protocol();
sprotocol.setName("SNMP");
dao.insert(sprotocol);



List<Protocol> pList = dao.getAll();
Protocol protocol = pList.get(0);



Protocol protocol2 = dao.getById(protocol.getId());
Assert.assertEquals("SNMP", protocol2.getName()); }

                                                    }
```

The corresponding data access method is executed using Hibernate 4 Session. get () operation.

```
public class ProtocolDaoImpl  implements ProtocolDao {
@Override
public Protocol getById(Integer id) {
return (Protocol) sessionFactory.getCurrentSession().get(Protocol.class, id); }

                                                    }
```

We will look at the delete operation writing the test first, followed by the data access operation.

```
public class ProtocolDaoImplTest {
@Test
public void testDelete() {
```

```
public void testDelete() {
Protocol sprotocol = new Protocol();
sprotocol.setName(“SNMP”);
dao.insert(sprotocol);


Protocol tProtocol = new Protocol();
tProtocol.setName(“TCP/IP”);
dao.insert(tProtocol);


Assert.assertEquals(2L, dao.getAll().size());


dao.delete(tProtocol);


Assert.assertEquals(1L, dao.getAll().size());
}
```

}

The delete data access operation uses Hibernate 4 Session.delete () operation.

```
public class ProtocolDaoImpl  implements ProtocolDao {
@Override
public void delete(Protocol protocol) {
sessionFactory.getCurrentSession().delete(protocol); }
```

}

Last we will look at the update operation.

```
public class ProtocolDaoImplTest {
@Test
public void testUpdate() {
Protocol sprotocol = new Protocol();
sprotocol.setName(“SNMP”);
dao.insert(sprotocol);
```

```
Assert.assertEquals(1L, dao.getAll().size());

List<Protocol> pList = dao.getAll();
Protocol protocol = pList.get(0);
protocol.setName("TCP/IP");


dao.update(protocol);


List<Protocol> pList2 = dao.getAll();
Protocol protocol2 = pList2.get(0);
Assert.assertEquals("TCP/IP", protocol2.getName()); }

                                                    }
```

The update data access operation is shown below which uses Hibernate 4 Session.merge () method.

```
public class ProtocolDaoImpl  implements ProtocolDao {
@Override
public void update(Protocol protocol) {
sessionFactory.getCurrentSession().merge(protocol); }

                                                    }
```

Develop the business service tier

We will lead off with the test for fetching all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class ProtocolServiceImplTest {
@Autowired
private ProtocolService service;
@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size()); }

                                                    }
```

}

Next, let us look at how to code the corresponding find all records method in the business service.

```
@Service
@Transactional
public class ProtocolServiceImpl implements ProtocolService {
@Autowired
private ProtocolDao protocolDao;
@Autowired
private ProtocolMapper protocolMapper;
@Override
public List<ProtocolDto> findAll() {
List<Protocol> protocols = protocolDao.getAll(); List<ProtocolDto>
protocolDtos = new ArrayList<ProtocolDto>(); for(Protocol protocol :
protocols){
protocolDtos.add(protocolMapper.mapEntityToDto(protocol)); }
return protocolDtos;

}

}
```

The mapper typically has two important operations which are used to convert data access objects to entities and vice versa.

```
@Component
public class ProtocolMapper {

public Protocol mapDtoToEntity(ProtocolDto protocolDto){
if(protocolDto instanceof TCPDto) {
TCP tcp = new TCP();
if(null!=protocolDto.getId()) tcp.setId(protocolDto.getId());
if(null!=protocolDto.getName()) tcp.setName(protocolDto.getName()); return
tcp;
} else if(protocolDto instanceof SNMPDto) {
SNMP snmp = new SNMP();
if(null!=protocolDto.getId()) snmp.setId(protocolDto.getId());
if(null!=protocolDto.getName()) snmp.setName(protocolDto.getName()); return
snmp;

}
```

}

return null;

}

public ProtocolDto mapEntityToDto(Protocol protocol){
if(protocol instanceof TCP) {
TCPDto tcpDto = new TCPDto();
if(null!=protocol.getId()) tcpDto.setId(protocol.getId());
if(null!=protocol.getName()) tcpDto.setName(protocol.getName()); return
tcpDto;
} else if(protocol instanceof SNMP) {
SNMPDto snmpDto = new SNMPDto();
if(null!=protocol.getId()) snmpDto.setId(protocol.getId());
if(null!=protocol.getName()) snmpDto.setName(protocol.getName()); return
snmpDto;

}

return null;

}

}

The create test operation is next which is used to create a new row in the database.
public class ProtocolServiceImplTest {
@Test
public void testCreate() {
ProtocolDto sprotocol = new SNMPDto();
sprotocol.setName(“SNMP”);
service.create(sprotocol);

ProtocolDto hProtocol = new TCPDto();
hProtocol.setName(“TCP/IP”);
service.create(hProtocol);

Assert.assertEquals(2L, service.findAll().size()); }

}

The corresponding method in the business service tier accepts a data access object, converts to entity, and delegates to the data access tier to commit to the database.

```
public class ProtocolServiceImpl implements ProtocolService {
@Override
public void create(ProtocolDto protocolDto) {
protocolDao.insert(protocolMapper.mapDtoToEntity(protocolDto)); }
```

}

The next operation is finding a row with a given identifier.

```
public class ProtocolServiceImplTest {
@Test
public void testFindById() {
ProtocolDto sprotocol = new SNMPDto();
sprotocol.setName(“SNMP”);
service.create(sprotocol);


List<ProtocolDto> pList = service.findAll(); ProtocolDto protocol =
pList.get(0);


ProtocolDto protocol2 = service.findById(protocol.getId());
Assert.assertEquals(“SNMP”, protocol2.getName()); }
```

}

We then describe the corresponding operation of business service to find a row with a given identifier.

```
public class ProtocolServiceImpl implements ProtocolService {
@Override
public ProtocolDto findById(Integer id) {
Protocol protocol = protocolDao.getById(id);
```

```
Protocol protocol = protocolDao.getById(id);
ProtocolDto protocolDto = null;
if(null !=protocol){
protocolDto = protocolMapper.mapEntityToDto(protocol); }
return protocolDto;
}

                                                    }
```

Next is the remove operation, which accepts a given identifier and removes the matching row from the database.

```
public class ProtocolServiceImplTest {
@Test
public void testRemove() {
ProtocolDto sprotocol = new SNMPDto();
sprotocol.setName("SNMP");
service.create(sprotocol);


ProtocolDto hProtocol = new TCPDto();
hProtocol.setName("TCP/IP");
service.create(hProtocol);


Assert.assertEquals(2L, service.findAll().size());
List<ProtocolDto> pList = service.findAll(); ProtocolDto protocol =
pList.get(0);
service.remove(protocol.getId());


Assert.assertEquals(1L, service.findAll().size()); }

                                                    }
```

The matching business service method is defined next which accepts an identifier and removes the matching instance from the database via the data access tier operation.

```
public class ProtocolServiceImpl implements ProtocolService {
@Override
```

```
public void remove(Integer protocolId) {
Protocol protocol = protocolDao.getById(protocolId);
protocolDao.delete(protocol);
}
}
```

The last test operation is the edit operation which takes a data access object and applies the data access method to merge the changes into the database.

```
public class ProtocolServiceImplTest {
@Test
public void testEdit() {
ProtocolDto sprotocol = new SNMPDto();
sprotocol.setName(“SNMP”);
service.create(sprotocol);



Assert.assertEquals(1L, service.findAll().size());
List<ProtocolDto> pList = service.findAll(); ProtocolDto protocol =
pList.get(0);
protocol.setName(“SNMP1”);
service.edit(protocol);



List<ProtocolDto> pList2 = service.findAll(); ProtocolDto protocol2 =
pList2.get(0);
Assert.assertEquals(“SNMP1”, protocol2.getName()); }

                                                    }
```

The equivalent operation of the business service is the edit operation described below.

```
public class ProtocolServiceImpl implements ProtocolService {
@Override
public void edit(ProtocolDto protocolDto) {
protocolDao.update(protocolMapper.mapDtoToEntity(protocolDto)); }

                                                    }
```

Develop the presentation tier

The last tier in the stack is the REST based Spring Controller. First, we will look at how to write the test and then follow with the corresponding method in the controller.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true) @Transactional
public class ProtocolControllerTest {

private Gson gson = new GsonBuilder().create();
@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc = MockMvcBuilders.
<StandaloneMockMvcBuilder>webAppContextSetup
(webApplicationContext).build();

                                                    }

@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                                    }


public void testCreate() throws Exception {
ProtocolDto protocolDto = new TCPDto();
protocolDto.setName(“ABC”);



String json = gson.toJson(protocolDto);
json= json.replace(“{“, “”);
json= json.replace(“}”, “”);
```

json = ¨{\´´type\´´:\´´tcp\´´,´´+json+´´}´´;

MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“*singletableinheritance*create”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}

public void testUpdate() throws Exception {

MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get(“*singletableinheritance*findAll”); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString(); Type listType = new
TypeToken<ProtocolDto[]>() {}.getType(); ProtocolDto[] protocolDtoList =
gson.fromJson(response2, listType); ProtocolDto protocolDto2 =
protocolDtoList[0]; protocolDto2.setName(“DEF”);
String json2 = gson.toJson(protocolDto2);
json2= json2.replace(“{“, “”);
json2= json2.replace(“}”, “”);
json2 = “{\”type\”:\”tcp\”,”+json2+”}”;
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post(“*singletableinheritance*edit”);
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

public void testDelete() throws Exception {

MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get(“*singletableinheritance*findAll”); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2

= result.getResponse().getContentAsString(); Type listType = new TypeToken<ProtocolDto[]>() {}.getType(); ProtocolDto[] protocolDtoList = gson.fromJson(response2, listType); ProtocolDto protocolDto2 = protocolDtoList[0]; MockHttpServletRequestBuilder requestBuilder3 = MockMvcRequestBuilders.post("*singletableinheritance*remove/"+protocolDto2.g requestBuilder3.contentType(MediaType.APPLICATION_JSON); this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st }

}

The agreeing REST, Spring Controller, will be contacting the business service tier to connect to the database. Please note that the code is similar to the mock controller developed earlier with the difference in the request mapping and the call going to the actual database instead of the mock in-memory database.

```
@Controller
@RequestMapping(value="/singletableinheritance") @Transactional
public class ProtocolController {

@Autowired
private ProtocolService service ;


@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody ProtocolDto[] findAll(){
List<ProtocolDto> list = service.findAll(); return list.toArray(new
ProtocolDto[list.size()]); }
@RequestMapping(value="/findById/{protocolid}",
method=RequestMethod.GET) public @ResponseBody ProtocolDto
findById(@PathVariable("protocolid") Integer protocolid){
return service.findById(protocolid);

}

@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody ProtocolDto protocol){
service.create(protocol);

}
```

}

@RequestMapping(value="/remove/{protocolid}",
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable("protocolid") Integer protocolid){
service.remove(protocolid);

}

@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody ProtocolDto protocol){
service.edit(protocol);

}

}

We have come to the end of the chapter where we will discuss how to integrate the development work done by both the teams. The REST service will need to be integrated with the actual user interface. Please note that the request mapping changes from "*singletableinheritance*mock" to "/singletableinheritance". The rest of the work involves creating the actual user interface by taking most of the contents of the mock user interface.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to single table inheritance
- Develop data access tier
- Develop business service tier

- Develop a REST service controller.

Chapter 15. Concrete Table Inheritance

In the previous chapter, we looked at the first approach to inheritance using a discriminator in a single table. In this chapter, we will look at the second approach to inheritance. This chapter is a briefing on the process of development with a concrete table inheritance relationship. We will first create the user interface and then write the actual service. At the end, we will join the two solutions and come to a judicious conclusion.

Domain Model

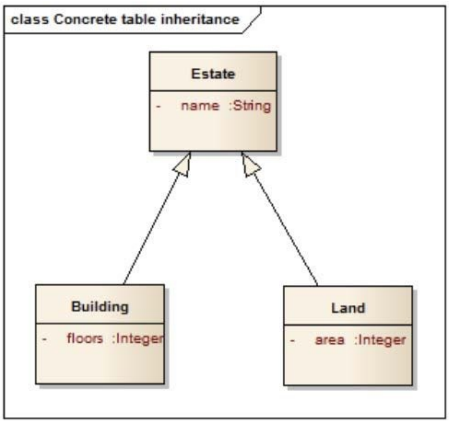

Figure 15-1: A Concrete Table Inheritance relationship We will begin with a

concrete table inheritance relationship, as shown in Figure 15-1. The Estate entity has an identifier and a name. The Building entity has the identifier and the name of the Estate along with a number of floors. The Land entity also has all the fields of Estate with the area of the land. The characteristic of the concrete table is that all classes have tables and the sub-types have all the fields of the super-type.

Develop a User Interface

We will divide the user interface development into three main tasks – the data transfer object, the mock service and the mock user interface. We will see each of these tasks one after another.

Develop the data transfer object

The data transfer object contains EstateDto which has an identifier and a name. There are two sub-types of EstateDto known as BuildingDto which also has a number of floors and LandDto which has an area.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include =
JsonTypeInfo.As.PROPERTY, property = “type”) @JsonSubTypes({
   @Type(value = BuildingDto.class, name = “building”), @Type(value =
LandDto.class, name = “land”) }) public class EstateDto {
private Integer id;
private String name;

  // getters and setters
 }
public class BuildingDto extends EstateDto {
@JsonProperty(“floors”)
private Integer floors;

  // getters and setters

                                                    }

public class LandDto extends EstateDto {
@JsonProperty(“area”)
private Integer area;
```

```
  // getters and setters

                                                    }
```

Develop the mock service

Let’s jump with a simple add to the in-memory convention database which is based on the Singleton pattern implemented as an enum.

```
public enum EstateInMemoryDB {

INSTANCE;



private static List<EstateDto> list = new ArrayList<EstateDto>(); private static
Integer lastId = 0;



public Integer getId() {
return ++lastId;

                                                    }

public void add(EstateDto estateDto) {
estateDto.setId(getId());
list.add(estateDto);

                                                    }

                                                    }
```

We will be watching the mock service for create functionality which uses the above in-memory catalogue for persisting the data.

```
@Controller
@RequestMapping(value=”concretetableinheritancemock”) public class
EstateMockController {
@RequestMapping(value=”/create”, method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody EstateDto estate){
```

EstateInMemoryDB.INSTANCE.add(estate);

}

}

Next we will air the method which is accountable for the retrieval of all the rows. First, we will review the in-memory database operations which the controller is going to use.
public enum EstateInMemoryDB {
public List<EstateDto> findAll() {
return list;

}

public EstateDto findById(Integer id) {
for (EstateDto dto:list) {
if (dto.getId()==id)
return dto;

}

return null;

}

}

The *findAll returns all instances of sub-types of EstateDto, and* findById returns a specific row of either LandDto or BuildingDto.
public class EstateMockController {
@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public @ResponseBody EstateDto[] findAll(){
List<EstateDto> list = EstateInMemoryDB.INSTANCE.findAll(); return list.toArray(new EstateDto[list.size()]); }
@RequestMapping(value=”/findById/{estateid}”, method=RequestMethod.GET) public @ResponseBody EstateDto findById(@PathVariable(“estateid”) Integer estateid){
return EstateInMemoryDB.INSTANCE.findById(estateid); }

}

Next we will see the remove operation which eliminates a specific row with an identifier.

```
public enum EstateInMemoryDB {
public void remove(Integer id) {
EstateDto toRemove = null;
for (EstateDto dto:list)
if (dto.getId()==id) toRemove = dto;
if (toRemove!=null) list.remove(toRemove);
}

}
```

The /remove operation deletes a specific row with the given identifier.

```
public class EstateMockController {
@RequestMapping(value=”/remove/{estateid}”,
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“estateid”) Integer estateid){
EstateInMemoryDB.INSTANCE.remove(estateid);
}

}
```

The edit method updates a specific row with the specifics. Let’s look at the in-memory database operations which the controller is going to use.

```
public class EstateInMemoryDB {
public void edit(EstateDto estateDto) {
for (EstateDto dto:list) {
if (dto.getId()==estateDto.getId()) {
dto.setName(estateDto.getName());
if(estateDto instanceof BuildingDto) {
((BuildingDto)dto).setFloors(((BuildingDto)estateDto).getFloors()); } else
if(estateDto instanceof LandDto) {
((LandDto)dto).setArea(((LandDto)estateDto).getArea()); }

}

}
```

}

}

The /edit operation updates a row with details using HTTP POST method.

```
public class EstateMockController {
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody EstateDto estate){
EstateInMemoryDB.INSTANCE.edit(estate);
}
```

}

Develop the mock user interface

Let’s now see how to produce the user interface shown in Figure 15-2.

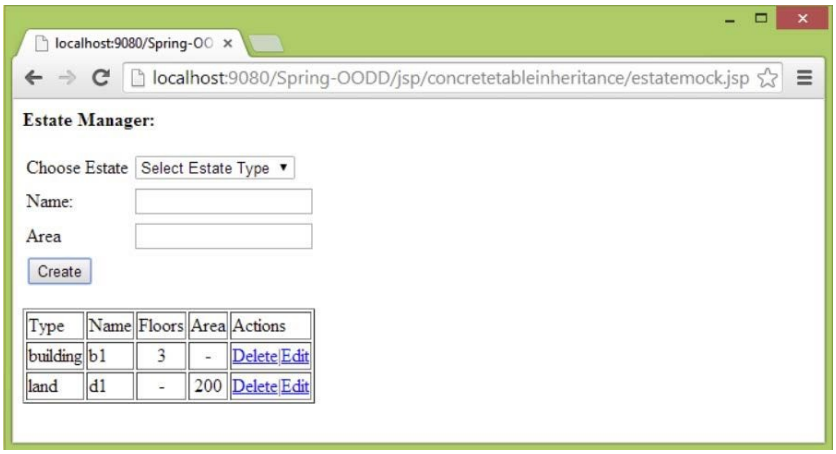

Figure 15-2: A screenshot of the Estate manager user interface Let’s first review the JSP fragment which constitutes the body of the page.

```
<body onload=”loadObjects()”>
<div id=”container”>
<div>
<p><b>Estate Manager:</b></p> </div>
<form method=”post” id=”estateForm”>
<table>
<tbody>
```

<tr>
<td>Choose Estate</td>
<td>
<select name=”estateChooser” id=”estateChooser”
onchange=”setEstateValue()”> <option value=””>Select Estate Type</option>
<option value=”building”>Building</option> <option
value=”land”>Land</option> </select>
</td>
</tr>
<tr>
<td>Name:</td>
<td>
<input name=”estateid” id=”estateid” type=”hidden”> <input
name=”estatename” id=”estatename” type=”text”> </td>
</tr>
<tr>
<td> Enter Value </td> <td>
<input name=”estateProperty” id=”estateProperty” type=”text”> </td>
</tr>
<tr>
<td colspan=”2”><input type=submit value=”Create” id=”subButton”
onclick=”return methodCall()”></td> </tr>
</tbody>
</table>
</form>

</body>
First we will see the JavaScript method which is called when the page is loaded with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDconcretetableinheritance/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)
                                                {
processResponseData(data);
}.
```

```
});
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("estatename").value="";
document.getElementById("estateProperty").value=""; alert("Error Status Load
Objects:"+textStatus); }

});

$('#estateChooser').get(0).selectedIndex = 0;
return false;

}
```

The JavaScript processResponseData method is called when the above service call is successful. Also the Javascript method generateTableData is called to form the table grid after the values are retrieved.

```
function processResponseData(responsedata){
var dyanamicTableRow="<table border=1>"+
"<tr>" +
"<td>Type</td>"+
"<td>Name</td>"+
"<td>Floors</td>"+
"<td>Area</td>"+
"<td>Actions</td>"+
"</tr>";
var dataRow="";
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue);

});

dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("estateFormResponse").innerHTML=dyanamicTable
}
function generateTableData(itemvalue){
var floorValue = (itemvalue.floors != "" && itemvalue.floors != null) ?
itemvalue.floors : "-"; var areaValue = (itemvalue.area != "" && itemvalue.area
!= null) ? itemvalue.area : "-"; var dataRow="<tr>" +
"<td>" +itemvalue.type+"</td>"+
"<td>" +itemvalue.name+"</td>"+
"<td align='center'>" +floorValue+"</td>"+
```

“<td align=’center’>” +areaValue+”</td>”+
“<td>” +
“Delete”+
“|Edit”+
“</td>”+
“</tr>”;
return dataRow;
}
We have seen how the page loading works. Now let’s look at the create and update functionality called using the Javascript methodCall method.

```
function methodCall(){
var buttonValue = document.getElementById(“subButton”).value;
if(buttonValue==”Create”){
create();
}else if(buttonValue==”Update”){
update();

                                        }

return false;

                                        }
```

First, we will review the Javascript create method.

```
function create(){
var name = $(“#estatename”).val();
var type = $(“#estateChooser”).val();
var estateProperty = $(“#estateProperty”).val(); var formData = “”;
if(type == “” || type == null) {
alert(“Please choose Estate Type”);
return false;

                                        }

if(name == “” || name == null) {
alert(“Please enter Estate name”);
return false;

                                        }
```

```
if(type == “building”) {
formData={“type”:”building”,”name”:name,”floors”:estateProperty}; } else if
(type == “land”) {
formData={“type”:”land”,”name”:name,”area”:estateProperty}; }


$.ajax({
url : “Spring-OODDconcretetableinheritance/mock/create”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

                                                {

document.getElementById(“estatename”).value=””;
document.getElementById(“estateProperty”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById(“estatename”).value=””;
document.getElementById(“estateProperty”).value=””; alert(“Error Status
Create:”+textStatus);

                                                }

                                               });

return false;

                                                }
```

Next we will foresee the Javascript update method.

```
function update(){
var name = $(“#estatename”).val();
var id = +$(“#estateid”).val();
var type = $(“#estateChooser”).val();
var estateProperty = $(“#estateProperty”).val();
if(type == “building”) {
```

```
if(type == "building") {
formData={"type":"building","id":id,"name":name,"floors":estateProperty}; }
else if (type == "land") {
formData={"type":"land","id":id,"name":name,"area":estateProperty}; }


$.ajax({
url : "Spring-OODDconcretetableinheritance/mock/edit", type: "POST",
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader("Accept", "application/json");
xhr.setRequestHeader("Content-Type", "application/json"); },
success: function(data, textStatus, jqXHR)

{

document.getElementById("estatename").value="";
document.getElementById("estateProperty").value="";
document.getElementById("subButton").value="Create"; loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
document.getElementById("estatename").value="";
document.getElementById("estateProperty").value=""; alert("Error Status
Update:"+textStatus);

}

});

return false;

}
```

The editObject and the viewObject Javascript method call allows the selection of a record for an update operation.

```
function editObject(estateid){
var editurl="Spring-OODDconcretetableinheritance/mock/findById/"+estateid;
var estateForm={id:estateid};
$.ajax({
url : editurl,
type: "GET",
```

type: "GET",
data : estateForm,
dataType: "json",
success: function(data, textStatus, jqXHR)

{

viewObject(data);
document.getElementById("subButton").value="Update";
},
error: function (jqXHR, textStatus, errorThrown) {
alert("Error Status Find Object:"+textStatus); }

});

}

function viewObject(data){
$("#estateChooser").val(data.type);
document.getElementById("estatename").value=data.name;
document.getElementById("estateid").value=data.id; if(data.type == "building")
{
document.getElementById("estateProperty").value=data.floors; } else
if(data.type == "land") {
document.getElementById("estateProperty").value=data.area; }

}

Lastly, we will air the Javascript deleteObject method.
function deleteObject(estateid){
var estateForm={id:estateid};
var delurl="*Spring-OODD*concretetableinheritance/mock/remove/"+estateid;
$.ajax({
url : delurl,
type: "POST",
data : estateForm,
dataType: "json",
success: function(data, textStatus, jqXHR)

{

```
loadObjects();
},
error: function (jqXHR, textStatus, errorThrown) {
alert("Error Status Delete:"+textStatus);

}

});

}
```

Develop the Service

The service development will be spread across four chief tasks – creating the entity, creating the data access tier, creating the business service tier and creating the JSON based REST service which will be integrated with the user interface. Let's look at one at a time and go bottom up starting with the entity and going up to the REST service.

Develop the resource (entity) tier First, we will look at the test for Estate entity which constitutes basic CRUD operations.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class EstateTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings("unchecked")
@Test
public void testCRUD()

{

Estate p1 = new Estate();
p1.setName("A");
```

p1.setName(“A”);

Estate p2 = new Estate();
p2.setName(“B”);

sessionFactory.getCurrentSession().save(p1);
sessionFactory.getCurrentSession().save(p2);

p1.setName(“C”);
sessionFactory.getCurrentSession().merge(p1);

List<Estate> list = sessionFactory.getCurrentSession().createQuery(“from Estate”).list(); Assert.assertEquals(2L, list.size());

sessionFactory.getCurrentSession().delete(p1);

List<Estate> list2 = sessionFactory.getCurrentSession().createQuery(“from Estate”).list(); Assert.assertEquals(1L, list2.size());

}

}

We will look at the definition of the Estate entity. The @Inheritance annotation defines the inheritance strategy which is a Concrete table inheritance also called Table per class. This strategy has a table for all the classes in the hierarchy and has all the fields of the parent classes in the child class. For example, the Land entity will have an identifier and the name of the parent entity and also an area field. Similarly, the Building entity will have an identifier and the name of the parent along with the floors field.

```
@Entity
@Table(name = “ESTATE”)
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) public class
```

```
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) public class
Estate {

private Integer id;
private String name;

@Id
@GeneratedValue(strategy = GenerationType.TABLE) @Column(name = “ID”)
public Integer getId() {
return id;

                                        }


public void setId(Integer id) {
this.id = id;

                                        }


@Column(name = “NAME”, nullable = false, length = 100) public String
getName() {
return name;

                                        }


public void setName(String name) {
this.name = name;

                                        }

                                        }
```

Next, we will look at the test for the Building entity.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class BuildingTest {
@Autowired
private SessionFactory sessionFactory;
```

```
@SuppressWarnings("unchecked")
@Test
public void testCRUD()
                                                {

Estate p1 = new Estate();
p1.setName("Estate");
sessionFactory.getCurrentSession().save(p1);


Building building = new Building();
building.setName("Building");
building.setFloors(20);
sessionFactory.getCurrentSession().save(building);
building.setName("Building Estate");
building.setFloors(30);
sessionFactory.getCurrentSession().merge(building);
List<Building> list = sessionFactory.getCurrentSession().createQuery("from
Building").list(); Assert.assertEquals(1L, list.size());


sessionFactory.getCurrentSession().delete(building);
List<Building> list2 = sessionFactory.getCurrentSession().createQuery("from
Building").list(); Assert.assertEquals(0L, list2.size());

                                                }

                                                }
```

We review the definition of the Building entity based on the test above.

```
@Entity
@Table(name="BUILDING")
@AttributeOverrides({
   @AttributeOverride(name="id", column=@Column(name="ID")),
   @AttributeOverride(name="name", column=@Column(name="NAME"))
})
public class Building extends Estate {
```

```
private Integer floors;

@Column(name = “FLOORS”, nullable = false, length = 100) public Integer
getFloors() {
return floors;

                                                    }


public void setFloors(Integer floors) {
this.floors = floors;

                                                    }

                                                    }
```

Lastly, we foresee the test for the Land entity which is similar to the test for the Building entity.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class LandTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
public void testCRUD()

                                                    {

Estate p1 = new Estate();
p1.setName(“Estate”);
sessionFactory.getCurrentSession().save(p1);


Land land = new Land();
land.setName(“Land”);
```

```
land.setArea(20);
sessionFactory.getCurrentSession().save(land);


land.setName("Land Estate");
land.setArea(30);
sessionFactory.getCurrentSession().merge(land);
List<Land> list = sessionFactory.getCurrentSession().createQuery("from
Land").list(); Assert.assertEquals(1L, list.size());


sessionFactory.getCurrentSession().delete(land);
List<Land> list2 = sessionFactory.getCurrentSession().createQuery("from
Land").list(); Assert.assertEquals(0L, list2.size());

                                                    }

                                                    }
```

Finally, we look at the Land entity definition.

```
@Entity
@Table(name="LAND")
@AttributeOverrides({
   @AttributeOverride(name="id", column=@Column(name="ID")),
   @AttributeOverride(name="name", column=@Column(name="NAME"))
})
public class Land extends Estate {


private Integer area;

@Column(name = "AREA", nullable = false, length = 100) public Integer
getArea() {
return area;

                                                    }


public void setArea(Integer area) {
```

```
public void setArea(Integer area) {
this.area = area;

                                                    }

}
```

Table structure

```
DROP TABLE `land`;
DROP TABLE `building`;
DROP TABLE `estate`;
CREATE TABLE `estate` (
  `ID` int(11) NOT NULL,
  `NAME` varchar(100) NOT NULL,
  PRIMARY KEY (`ID`)

                                                    ) ;

CREATE TABLE `building` (
  `ID` int(11) NOT NULL,
  `NAME` varchar(100) NOT NULL,
  `FLOORS` int(11) NOT NULL,
  PRIMARY KEY (`ID`)

                                                    ) ;

CREATE TABLE `land` (
  `ID` int(11) NOT NULL,
  `NAME` varchar(100) NOT NULL,
  `AREA` int(11) NOT NULL,
  PRIMARY KEY (`ID`)

                                                    ) ;
```

Develop the data access tier

We will begin with the test case for the find all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
```

```
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class EstateDaoImplTest {
@Autowired
private EstateDao dao ;
@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size());

                                                    }

                                                    }
```

We will then see how to write the corresponding data access operation of the above test.

```
@Repository
@Transactional
public class EstateDaoImpl  implements EstateDao {
@Autowired
private SessionFactory sessionFactory ;
@SuppressWarnings("unchecked")
@Override
public List<Estate> getAll() {
return sessionFactory.getCurrentSession().createQuery("select estate from Estate
estate order by estate.id desc").list(); }

                                                    }
```

Let's move on to the insert operation test.

```
public class EstateDaoImplTest {
@Autowired
private EstateDao dao ;

@Test
public void testInsert() {
Estate testate = new Estate();
testate.setName("Royal Estate");
dao.insert(testate);
```

```
Estate sEstate = new Estate();
sEstate.setName("Majestic Estate");
dao.insert(sEstate);


Assert.assertEquals(2L, dao.getAll().size());

                                                    }

                                                    }
```

The corresponding insert operation in the data access class has to be implemented using Hibernate 4 Session. save () function.

```
public class EstateDaoImpl  implements EstateDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public void insert(Estate estate) {
sessionFactory.getCurrentSession().save(estate); }

                                                    }
```

We will jump to the next test operation which is finding a row with an identifier.

```
public class EstateDaoImplTest{
@Test
public void testGetById() {
Estate sestate = new Estate();
sestate.setName("Majestic Estate");
dao.insert(sestate);


List<Estate> pList = dao.getAll();
Estate estate = pList.get(0);


Estate estate2 = dao.getById(estate.getId());
Assert.assertEquals("Majestic Estate", estate2.getName()); }

                                                    }
```

}

The corresponding data access method is executed using Hibernate 4 Session. get () operation.

```
public class EstateDaoImpl  implements EstateDao {
@Override
public Estate getById(Integer id) {
return (Estate) sessionFactory.getCurrentSession().get(Estate.class, id); }
```

}

We will review the delete operation writing the test first, followed by the data access operation.

```
public class EstateDaoImplTest {
@Test
public void testDelete() {
Estate sestate = new Estate();
sestate.setName(“Majestic Estate”);
dao.insert(sestate);


Estate tEstate = new Estate();
tEstate.setName(“Royal Estate”);
dao.insert(tEstate);


Assert.assertEquals(2L, dao.getAll().size());


dao.delete(tEstate);


Assert.assertEquals(1L, dao.getAll().size());
```

}

}

The delete data access operation uses Hibernate 4 Session.delete () operation.

```
public class EstateDaoImpl  implements EstateDao {
@Override
public void delete(Estate estate) {
sessionFactory.getCurrentSession().delete(estate); }

                                                    }
```

Last we will see the update test operation.

```
public class EstateDaoImplTest{
@Test
public void testUpdate() {
Estate sestate = new Estate();
sestate.setName("Majestic Estate");
dao.insert(sestate);


Assert.assertEquals(1L, dao.getAll().size());

List<Estate> pList = dao.getAll();
Estate estate = pList.get(0);
estate.setName("Royal Estate");


dao.update(estate);


List<Estate> pList2 = dao.getAll();
Estate estate2 = pList2.get(0);
Assert.assertEquals("Royal Estate", estate2.getName()); }

                                                    }
```

The update data access operation is shown below which uses Hibernate 4 Session.merge () method.

```
public class EstateDaoImpl implements EstateDao {
@Override
public void update(Estate estate) {
sessionFactory.getCurrentSession().merge(estate); }
```

sessionFactory.getCurrentSession().merge(estate); }

}

Develop the business service tier

We will proceed with the test for the fetching all records function.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true ) @Transactional
public class EstateServiceImplTest {
@Autowired
private EstateService service;

@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size()); }
```

}

Let us look at how to code the corresponding find all records method in the business service.

```
@Service
@Transactional
public class EstateServiceImpl implements EstateService {

@Autowired
private EstateDao estateDao;

@Autowired
private EstateMapper estateMapper;

@Override
public List<EstateDto> findAll() {
List<Estate> estates = estateDao.getAll(); List<EstateDto> estateDtos = new
ArrayList<EstateDto>(); for(Estate estate : estates){
estateDtos.add(estateMapper.mapEntityToDto(estate)); }
return estateDtos;
```

}

}

The mapper characteristically has two vital operations which are used to convert data access objects to entities and vice versa.

```
@Component
public class EstateMapper {

public Estate mapDtoToEntity(EstateDto estateDto){
if(estateDto instanceof BuildingDto) {
Building building = new Building();
if(null!=estateDto.getId()) building.setId(estateDto.getId());
if(null!=estateDto.getName()) building.setName(estateDto.getName()); if(null!=
((BuildingDto)estateDto).getFloors())
building.setFloors(((BuildingDto)estateDto).getFloors()); return building;
} else if(estateDto instanceof LandDto) {
Land land = new Land();
if(null!=estateDto.getId()) land.setId(estateDto.getId());
if(null!=estateDto.getName()) land.setName(estateDto.getName()); if(null!=
((LandDto)estateDto).getArea()) land.setArea(((LandDto)estateDto).getArea());
return land;
```

}

```
return null;
```

}

```
public EstateDto mapEntityToDto(Estate estate){
if(estate instanceof Building) {
BuildingDto buildingDto = new BuildingDto();
if(null!=estate.getId()) buildingDto.setId(estate.getId());
if(null!=estate.getName()) buildingDto.setName(estate.getName()); if(null!=
((Building)estate).getFloors())
buildingDto.setFloors(((Building)estate).getFloors()); return buildingDto;
} else if(estate instanceof Land) {
LandDto landDto = new LandDto();
if(null!=estate.getId()) landDto.setId(estate.getId()); if(null!=estate.getName())
landDto.setName(estate.getName()); if(null!=((Land)estate).getArea())
landDto.setArea(((Land)estate).getArea()); return landDto;
```

```
                                                    }

return null;

                                                    }

                                                    }
```

The following create test operation is used to create a new row in the database.

```
public class EstateServiceImplTest {

@Test
public void testCreate() {
EstateDto bestate = new BuildingDto();
bestate.setName(“Majestic Estate”);
((BuildingDto)bestate).setFloors(10);
service.create(bestate);


EstateDto hEstate = new LandDto();
hEstate.setName(“Royal Estate”);
((LandDto)hEstate).setArea(200);
service.create(hEstate);


Assert.assertEquals(2L, service.findAll().size()); }

                                                    }
```

The matching method in the business service tier accepts a data access object, converts to entity and delegates to the data access tier to commit to the database.

```
public class EstateServiceImpl implements EstateService {
@Override
public void create(EstateDto estateDto) {
estateDao.insert(estateMapper.mapDtoToEntity(estateDto)); }

                                                    }
```

The next operation we will define is finding a row with a given identifier.

```
public class EstateServiceImplTest {
@Test
public void testFindById() {
EstateDto bestate = new BuildingDto();
bestate.setName(“Majestic Estate”);
((BuildingDto)bestate).setFloors(10);
service.create(bestate);


List<EstateDto> pList = service.findAll(); EstateDto estate = pList.get(0);


EstateDto estate2 = service.findById(estate.getId());
Assert.assertEquals(“Majestic Estate”, estate2.getName()); }

                                                    }
```

The equivalent operation of business service to find a row with a given identifier will now be labeled.

```
public class EstateServiceImpl implements EstateService {
@Override
public EstateDto findById(Integer id) {
Estate estate = estateDao.getById(id);
EstateDto estateDto = null;
if(null !=estate){
estateDto = estateMapper.mapEntityToDto(estate); }
return estateDto;
}

                                                    }
```

The remove procedure is then in line, which accepts a given identifier and removes the alike row from the database.

```
public class EstateServiceImplTest {
@Test
public void testRemove() {
EstateDto bestate = new BuildingDto();
bestate.setName(“Majestic Estate”);
```

```
((BuildingDto)bestate).setFloors(10);
service.create(bestate);


EstateDto hEstate = new LandDto();
hEstate.setName(“Royal Estate”);
((LandDto)hEstate).setArea(200);
service.create(hEstate);


Assert.assertEquals(2L, service.findAll().size());
List<EstateDto> pList = service.findAll(); EstateDto estate = pList.get(0);
service.remove(estate.getId());


Assert.assertEquals(1L, service.findAll().size()); }

                                                    }
```

The equivalent business service method is to be demarcated next which accepts an identifier and removes the matching instance from the database via the data access tier operation.

```
public class EstateServiceImpl implements EstateService {
@Override
public void remove(Integer estateId) {
Estate estate = estateDao.getById(estateId);
estateDao.delete(estate);
}
}
```

The last test operation we will describe is the edit operation which takes a data access object and applies the data access method to merge the changes into the database.

```
public class EstateServiceImplTest {
@Test
public void testEdit() {
EstateDto bestate = new BuildingDto();
bestate.setName(“Majestic Estate”);
((BuildingDto)bestate).setFloors(10);
service.create(bestate);
```

service.create(bestate);

Assert.assertEquals(1L, service.findAll().size());
List<EstateDto> pList = service.findAll(); EstateDto estate = pList.get(0);
estate.setName(“Royal Estate”);

service.edit(estate);

List<EstateDto> pList2 = service.findAll(); EstateDto estate2 = pList2.get(0);
Assert.assertEquals(“Royal Estate”, estate2.getName()); }

}

The corresponding operation of the business service is the edit operation which is defined below.
public class EstateServiceImpl implements EstateService {
@Override
public void edit(EstateDto estateDto) {
estateDao.update(estateMapper.mapDtoToEntity(estateDto)); }

}

Develop the presentation tier

The last level in the stack is the REST based Spring Controller. First, we will look at how to write the test and then follow with the corresponding controller.
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration(locations = { “classpath:context.xml” })
@TransactionConfiguration(defaultRollback = true) @Transactional
public class EstateControllerTest {

private Gson gson = new GsonBuilder().create();

@Resource
private WebApplicationContext webApplicationContext;

```
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc = MockMvcBuilders.
<StandaloneMockMvcBuilder>webAppContextSetup
(webApplicationContext).build();

                                        }

@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                        }


public void testCreate() throws Exception {



EstateDto estateDto = new BuildingDto();
estateDto.setName(“ABC”);
((BuildingDto)estateDto).setFloors(20);
String json = gson.toJson(estateDto);
json= json.replace(“{“, “”);
json= json.replace(“}”, “”);
json = “{\”type\”:\”building\”,”+json+”}”;
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post(“concretetableinheritancecreate”);
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}


public void testUpdate() throws Exception {
```

```
MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("concretetableinheritancefindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString();
Type listType = new TypeToken<EstateDto[]>() {}.getType(); EstateDto[]
estateDtoList = gson.fromJson(response2, listType); EstateDto estateDto2 =
estateDtoList[0];


EstateDto newEstateDto = new BuildingDto();
newEstateDto.setId(estateDto2.getId());
newEstateDto.setName("DEF");
((BuildingDto)newEstateDto).setFloors(21);


String json2 = gson.toJson(newEstateDto);
json2= json2.replace("{", "");
json2= json2.replace("}", "");
json2 = "{\"type\":\"building\","+json2+"}";
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("concretetableinheritanceedit");
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

public void testDelete() throws Exception {


MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("concretetableinheritancefindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString();
Type listType = new TypeToken<EstateDto[]>() {}.getType(); EstateDto[]
estateDtoList = gson.fromJson(response2, listType); EstateDto estateDto2 =
estateDtoList[0];
```

```
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("concretetableinheritanceremove/"+estateDto2.g
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

                                    }
```

The matching REST Spring Controller will be contacting the business service tier to connect to the database. Please note that the code is like the mock controller developed earlier with the difference of the request mapping and the call going to the actual database instead of the mock in-memory database.

```
@Controller
@RequestMapping(value="/concretetableinheritance") @Transactional
public class EstateController {

@Autowired
private EstateService service;


@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody EstateDto[] findAll(){
List<EstateDto> list = service.findAll(); return list.toArray(new
EstateDto[list.size()]); }
@RequestMapping(value="/findById/{estateid}",
method=RequestMethod.GET) public @ResponseBody EstateDto
findById(@PathVariable("estateid") Integer estateid){
return service.findById(estateid);

                                    }

@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody EstateDto estate){
service.create(estate);

                                    }

@RequestMapping(value="/remove/{estateid}",
method=RequestMethod.POST) @ResponseStatus(value =
```

```
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable("estateid") Integer estateid){
service.remove(estateid);

                                                    }

@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody EstateDto estate){
service.edit(estate);

                                                    }

                                                    }
```

We have reached the end of the chapter where we will discuss how to integrate the development work done by both the teams. The REST service will need to be integrated with the actual user interface. Please note that the request mapping changes from "*concretetableinheritance*mock" to "/concretetableinheritance". The remaining work involves creating the actual user interface by taking most of the contents of the mock user interface.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to concrete table inheritance
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Chapter 16. Class Table Inheritance In the previous two chapters, we looked at the two approaches to inherit: one as a single table and the other as a concrete table. In this chapter, we will look at the third and final approach to inheritance. This chapter is a briefing on the process of development with a class table inheritance relationship. We will conclude our journey of creating the user interface at the start and then write the genuine service. At the conclusion, we will join both the solutions and come to a thoughtful inference.

Domain Model

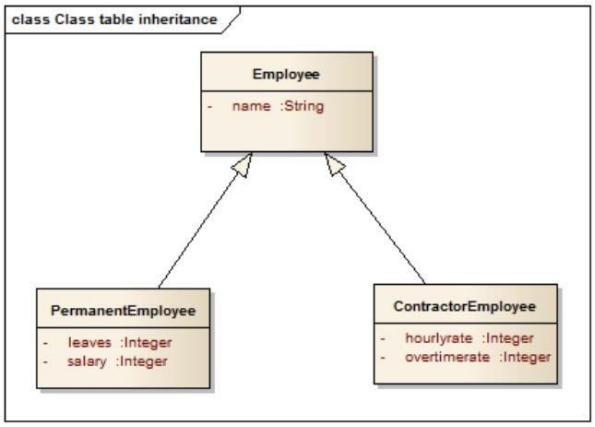

Figure 16-1: A Class Table Inheritance relationship

We will start with a class table inheritance relationship as shown in Figure 16-1. This is an example of a normalized design. The Employee entity has an identifier and a String name field. The Permanent entity has an identifier which is a foreign key to the parent and has fields such as leaves and salary. The Contractor entity also has an identifier which is a foreign key with the Employee identifier, and fields such as hourly rates and overtime rate.

Develop a User Interface

We will break the user interface development into three main tasks – the data transfer object, the mock service and the mock user interface. We will look at these tasks one after another.

Develop the data transfer object

The data transfer object contains EmployeeDto which has an identifier and a name. There are two sub-types of EmployeeDto known as PermanentEmployeeDto which also has salary and leaves, and ContractorEmployeeDto which has hourly rate and overtime rate.

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include =
JsonTypeInfo.As.PROPERTY, property = “type”) @JsonSubTypes({
   @Type(value = PermanentEmployeeDto.class, name =
“permanentEmployee”), @Type(value = ContractorEmployeeDto.class, name =
“contractorEmployee”) }) public class EmployeeDto {
private Integer id;
private String name;

// getters and setters
 }
public class PermanentEmployeeDto extends EmployeeDto {
@JsonProperty(“leaves”)
private Integer leaves;
@JsonProperty(“salary”)
private Integer salary;

  // getters and setters
```

}

```
public class ContractorEmployeeDto extends EmployeeDto {
@JsonProperty(“hourlyRate”)
private Integer hourlyRate;
@JsonProperty(“overtimeRate”)
private Integer overtimeRate;

  // getters and setters

}
```

Develop the mock service

Let’s begin with a simple add to the in-memory convention database which is based on the Singleton pattern implemented as an enum.

```
public enum EmployeeInMemoryDB {

INSTANCE;


private static List<EmployeeDto> list = new ArrayList<EmployeeDto>();
private static Integer lastId = 0;


public Integer getId() {
return ++lastId;

}

public void add(EmployeeDto employeeDto) {
employeeDto.setId(getId());
list.add(employeeDto);

}

}
```

We will be watching the mock service for create functionality which uses the above in-memory catalogue for persisting the data.

```
@Controller
@RequestMapping(value="classtableinheritancemock") public class
EmployeeMockController {
@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody EmployeeDto employee){
EmployeeInMemoryDB.INSTANCE.add(employee);

                                                    }

                                                    }
```

Next we will air the method which is accountable for the retrieval of all the rows. First, we will review the in-memory database operations which the controller is going to use.

```
public enum EmployeeInMemoryDB{
public List<EmployeeDto> findAll() {
return list;

                                                    }

public EmployeeDto findById(Integer id) {
for (EmployeeDto dto:list) {
if (dto.getId()==id)
return dto;

                                                    }

return null;

                                                    }

                                                    }
```

The *findAll returns all instances of sub-types of EmployeeDto and* findById returns a specific row of either PermanentDto or ContractorDto.

```
public class EmployeeMockController {
```

@RequestMapping(value=”/findAll”, method=RequestMethod.GET) public @ResponseBody EmployeeDto[] findAll(){
List<EmployeeDto> list = EmployeeInMemoryDB.INSTANCE.findAll(); return list.toArray(new EmployeeDto[list.size()]);

}

@RequestMapping(value=”/findById/{employeeid}”, method=RequestMethod.GET) public @ResponseBody EmployeeDto findById(@PathVariable(“employeeid”) Integer employeeid){
return EmployeeInMemoryDB.INSTANCE.findById(employeeid); }

}

Next we will see the remove operation which eliminates a specific row with an identifier.
public enum EmployeeInMemoryDB {
public void remove(Integer id) {
EmployeeDto toRemove = null;
for (EmployeeDto dto:list)
if (dto.getId()==id) toRemove = dto;
if (toRemove!=null) list.remove(toRemove);

}

}

The /remove operation deletes a specific row with the given identifier.
public class EmployeeMockController {
@RequestMapping(value=”/remove/{employeeid}”, method=RequestMethod.POST) @ResponseStatus(value = HttpStatus.NO_CONTENT)
public void remove(@PathVariable(“employeeid”) Integer employeeid){
EmployeeInMemoryDB.INSTANCE.remove(employeeid);

}

}

The edit method updates a specific row with the specifics. Let’s look at the in-memory database operations which the controller is going to use.

```
public class EmployeeInMemoryDB {
public void edit(EmployeeDto employeeDto) {
for (EmployeeDto dto:list) {
if (dto.getId()==employeeDto.getId()) {
dto.setName(employeeDto.getName());
if(employeeDto instanceof PermanentEmployeeDto) {
((PermanentEmployeeDto)dto).setLeaves(((PermanentEmployeeDto)employeeDt
((PermanentEmployeeDto)dto).setSalary(((PermanentEmployeeDto)employeeDt
} else if(employeeDto instanceof ContractorEmployeeDto) {
((ContractorEmployeeDto)dto).setHourlyRate(((ContractorEmployeeDto)employ
((ContractorEmployeeDto)dto).setOvertimeRate(((ContractorEmployeeDto)empl
}
                                                    }
                                                    }
                                                    }
                                                    }
```

The /edit operation updates a row with details using HTTP POST method.

```
public class EmployeeMockController {
@RequestMapping(value=”/edit”, method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody EmployeeDto employee){
EmployeeInMemoryDB.INSTANCE.edit(employee);
                                                    }
                                                    }
```

Develop the mock user interface

Let’s now see how to produce the user interface shown in Figure 16-2.

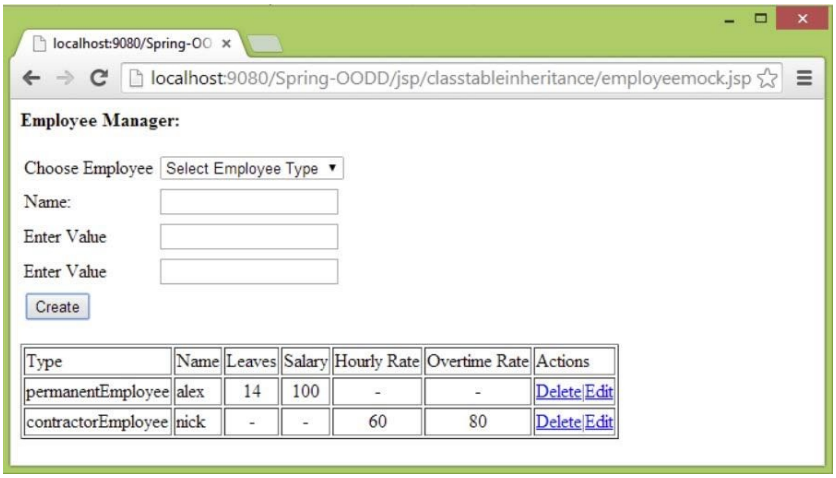

Figure 16-2: A screenshot of the Employee Manager user interface Let's first review the JSP fragment which constitutes the body of the page.

```
<body onload="loadObjects()">
<div id="container">
<div>
<p><b>Employee Manager:</b></p> </div>
<form method="post" id="employeeForm">
<table>
<tbody>
<tr>
<td>Choose Employee</td>
<td>
<select name="employeeChooser" id="employeeChooser"
onchange="setEmployeeValue()"> <option value="">Select Employee
Type</option> <option
value="permanentEmployee">PermanentEmployee</option> <option
value="contractorEmployee">ContractorEmployee</option> </select>
</td>
</tr>
<tr>
<td>Name:</td>
<td>
<input name="employeeid" id="employeeid" type="hidden"> <input
name="employeename" id="employeename" type="text"> </td>
</tr>
<tr>
<td><div id="employeeFirstValue">Enter Value</div></td> <td>
```

<input name=”employeeFirstProperty” id=”employeeFirstProperty” type=”text”> </td>
</tr>
<tr>
<td> Enter Value </td> <td>
<input name=”employeeSecondProperty” id=”employeeSecondProperty” type=”text”> </td>
</tr>
<tr>
<td colspan=”2”><input type=submit value=”Create” id=”subButton” onclick=”return methodCall()”></td> </tr>
</tbody>
</table>
</form>

</body>
First we will see the JavaScript method which is called when the page is loaded with the onload event.

```
function loadObjects(){
$.ajax({
url : “Spring-OODDclasstableinheritance/mock/findAll”, type: “GET”,
data : {},
dataType: “json”,
success: function(data, textStatus, jqXHR)
{
processResponseData(data);
},
error: function (jqXHR, textStatus, errorThrown)
{
document.getElementById(“employeename”).value=””;
document.getElementById(“employeeFirstProperty”).value=””;
document.getElementById(“employeeSecondProperty”).value=””; alert(“Error Status Load Objects:”+textStatus);
}
```

}

});

$('#employeeChooser').get(0).selectedIndex = 0;
return false;

}

The JavaScript processResponseData method is called when the above service call is successful. Also the Javascript method generateTableData is called to form the table grid after the values are retrieved.

```
function processResponseData(responsedata){
var dyanamicTableRow="<table border=1>"+
"<tr>" +
"<td>Type</td>"+
"<td>Name</td>"+
"<td>Leaves</td>"+
"<td>Salary</td>"+
"<td>Hourly Rate</td>"+
"<td>Overtime Rate</td>"+
"<td>Actions</td>"+
"</tr>";
var dataRow="";
$.each(responsedata, function(itemno, itemvalue){
dataRow=dataRow+generateTableData(itemvalue);

});

dyanamicTableRow=dyanamicTableRow+dataRow+"</table>";
document.getElementById("employeeFormResponse").innerHTML=dyanamicTa
}
function generateTableData(itemvalue){
var leaves = (itemvalue.leaves != "" && itemvalue.leaves != null) ?
itemvalue.leaves : "-"; var salary = (itemvalue.salary != "" && itemvalue.salary
!= null) ? itemvalue.salary : "-"; var hourlyRate = (itemvalue.hourlyRate != ""
&& itemvalue.hourlyRate != null) ? itemvalue.hourlyRate : "-"; var
overtimeRate = (itemvalue.overtimeRate != "" && itemvalue.overtimeRate !=
null) ? itemvalue.overtimeRate : "-";
var dataRow="<tr>" +
"<td>" +itemvalue.type+"</td>"+
```

“<td>” +itemvalue.type+”</td>” +
“<td>” +itemvalue.name+”</td>”+
“<td align=’center’>” +leaves+”</td>”+
“<td align=’center’>” +salary+”</td>”+
“<td align=’center’>” +hourlyRate+”</td>”+
“<td align=’center’>” +overtimeRate+”</td>”+
“<td>” +
“Delete”+
“|Edit”+
“</td>”+
“</tr>”;
return dataRow;
}

We have seen how the page loading works. Now let’s look at the create and update functionality called using the Javascript methodCall method.

```
function methodCall(){
var buttonValue = document.getElementById(“subButton”).value;
if(buttonValue==”Create”){
create();
}else if(buttonValue==”Update”){
update();

                                                    }

return false;

                                                    }
```

First, we will review the Javascript create method.

```
function create(){
var name = $(“#employeename”).val();
var type = $(“#employeeChooser”).val();
var employeeFirstProperty = $(“#employeeFirstProperty”).val(); var
employeeSecondProperty = $(“#employeeSecondProperty”).val(); var formData
= “”;
if(type == “” || type == null) {
alert(“Please choose Employee Type”);
return false;

                                                    }
```

```
if(name == “” || name == null) {
alert(“Please enter Employee name”);
return false;

                                                }

if(type == “permanentEmployee”) {
formData={“type”:”permanentEmployee”,”name”:name,”leaves”:
employeeFirstProperty,”salary”:employeeSecondProperty}; } else if (type ==
“contractorEmployee”) {
formData={“type”:”contractorEmployee”,”name”:name,”hourlyRate”:
employeeFirstProperty,”overtimeRate”:employeeSecondProperty}; }
$.ajax({
url : “Spring-OODDclasstableinheritance/mock/create”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)

                                                {

document.getElementById(“employeename”).value=””;
document.getElementById(“employeeFirstProperty”).value=””;
document.getElementById(“employeeSecondProperty”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
setEmployeeValue();
},
error: function (jqXHR, textStatus, errorThrown)

                                                {

document.getElementById(“employeename”).value=””;
document.getElementById(“employeeFirstProperty”).value=””;
document.getElementById(“employeeSecondProperty”).value=””; alert(“Error
Status Create:”+textStatus);

                                                }

                                                });
```

return false;

}

Next we will foresee the Javascript update method.

```
function update(){
var name = $(“#employeename”).val();
var id = +$(“#employeeid”).val();
var type = $(“#employeeChooser”).val();
var employeeFirstProperty = $(“#employeeFirstProperty”).val(); var
employeeSecondProperty = $(“#employeeSecondProperty”).val();
if(type == “permanentEmployee”) {
formData={“type”:”permanentEmployee”,”id”:id,”name”:
name,”leaves”:employeeFirstProperty,”salary”:
employeeSecondProperty};
} else if (type == “contractorEmployee”) {
formData={“type”:”contractorEmployee”,”id”:id,”name”:name,”hourlyRate”:
employeeFirstProperty,”overtimeRate”:
employeeSecondProperty};
```

}

```
$.ajax({
url : “Spring-OODDclasstableinheritance/mock/edit”, type: “POST”,
data : JSON.stringify(formData),
beforeSend: function(xhr) {
xhr.setRequestHeader(“Accept”, “application/json”);
xhr.setRequestHeader(“Content-Type”, “application/json”); },
success: function(data, textStatus, jqXHR)
```

{

```
document.getElementById(“employeename”).value=””;
document.getElementById(“employeeFirstProperty”).value=””;
document.getElementById(“employeeSecondProperty”).value=””;
document.getElementById(“subButton”).value=”Create”; loadObjects();
setEmployeeValue();
},
error: function (jqXHR, textStatus, errorThrown)
```

{

document.getElementById(“employeename”).value=””;
document.getElementById(“employeeFirstProperty”).value=””;
document.getElementById(“employeeSecondProperty”).value=””; alert(“Error Status Update:”+textStatus);

}

});

return false;

}

The editObject and the viewObject Javascript method call allows the selection of a record for an update operation.
function editObject(employeeid){
var editurl=”*Spring-OODD*classtableinheritance/mock/findById/”+employeeid;
var employeeForm={id:employeeid};
$.ajax({
url : editurl,
type: “GET”,
data : employeeForm,
dataType: “json”,
success: function(data, textStatus, jqXHR)

{

viewObject(data);
document.getElementById(“subButton”).value=”Update”; setEmployeeValue();
},
error: function (jqXHR, textStatus, errorThrown)

{

alert(“Error Status Find Object:”+textStatus);

}

});

});

}

function viewObject(data){
$("#employeeChooser").val(data.type);
document.getElementById("employeename").value=data.name;
document.getElementById("employeeid").value=data.id; if(data.type ==
"permanentEmployee") {
document.getElementById("employeeFirstProperty").value=data.leaves;
document.getElementById("employeeSecondProperty").value=data.salary; }
else if(data.type == "contractorEmployee") {
document.getElementById("employeeFirstProperty").value=data.hourlyRate;
document.getElementById("employeeSecondProperty").value=data.overtimeRat
}

}

Lastly, we will air the Javascript deleteObject method.
function deleteObject(employeeid){
var employeeForm={id:employeeid};
var delurl="*Spring-OODD*classtableinheritance/mock/remove/"+employeeid;
$.ajax({
url : delurl,
type: "POST",
data : employeeForm,
dataType: "json",
success: function(data, textStatus, jqXHR)

{

loadObjects();
},
error: function (jqXHR, textStatus, errorThrown)

{

alert("Error Status Delete:"+textStatus);

}

});

}

Develop the Service

The service development will be spread across four principal tasks – creating the entity, creating the data access tier, creating the business service tier, and creating the JSON based REST service which will be integrated with the user interface. Let's look at one at a time and go bottom up, starting with the entity and going up to the REST service.

Develop the resource (entity) tier First, we will look at the test for Employee entity which constitutes basic CRUD operations.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { "classpath:context.xml" } )
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class EmployeeTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings("unchecked")
@Test
public void testCRUD()

{

Employee e1 = new Employee();
e1.setName("A");


Employee e2 = new Employee();
```

e2.setName(“B”);

sessionFactory.getCurrentSession().save(e1);
sessionFactory.getCurrentSession().save(e2);

e1.setName(“C”);
sessionFactory.getCurrentSession().merge(e1);

List<Employee> list = sessionFactory.getCurrentSession().createQuery(“from Employee”).list(); Assert.assertEquals(2L, list.size());

sessionFactory.getCurrentSession().delete(e1);

List<Employee> list2 = sessionFactory.getCurrentSession().createQuery(“from Employee”).list(); Assert.assertEquals(1L, list2.size());

}

}

We will look at the definition of the Employee entity. The @Inheritance annotation defines the inheritance strategy which is a Class table inheritance also called Joined strategy. This strategy has a table for all the classes in the hierarchy and is a normalized design. For example, the Permanent entity will have an identifier which is a foreign key to the Employee entity and the salary and leaves field. Similarly, the Contractor entity will have an identifier foreign key to the Employee and the hourly rate and overtime rate fields.

```
@Entity
@Table(name = “EMPLOYEE”)
@Inheritance(strategy = InheritanceType.JOINED)
public class Employee {

private Integer id;
private String name;
```

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE)
@Column(name = “ID”)
public Integer getId() {
return id;

                                                    }


public void setId(Integer id) {
this.id = id;

                                                    }


@Column(name = “NAME”, nullable = false, length = 100) public String
getName() {
return name;

                                                    }


public void setName(String name) {
this.name = name;

                                                    }

                                                    }
```

Next, we will look at the test for the PermanentEmployee entity.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class PermanentEmployeeTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
```

```
public void testCRUD()
                                        {

Employee e1 = new Employee();
e1.setName(“A”);
sessionFactory.getCurrentSession().save(e1);


PermanentEmployee permanentEmployee = new PermanentEmployee();
permanentEmployee.setName(“Lalit Narayan Mishra”);
permanentEmployee.setLeaves(20);
permanentEmployee.setSalary(500000);
sessionFactory.getCurrentSession().save(permanentEmployee);
permanentEmployee.setName(“Amritendu De”);
sessionFactory.getCurrentSession().merge(permanentEmployee);
List<PermanentEmployee> list =
sessionFactory.getCurrentSession().createQuery(“from
PermanentEmployee”).list(); Assert.assertEquals(1L, list.size());


sessionFactory.getCurrentSession().delete(permanentEmployee);
List<PermanentEmployee> list2 =
sessionFactory.getCurrentSession().createQuery(“from
PermanentEmployee”).list(); Assert.assertEquals(0L, list2.size());

                                        }

                                        }
```

We review the definition of the PermanentEmployee entity based on the test above.

```
@Entity
@Table(name=”PERMANENT_EMPLOYEE”)
@PrimaryKeyJoinColumn(name=”ID”)
public class PermanentEmployee extends Employee {


private Integer leaves;
```

```
private Integer salary;


@Column(name = “LEAVES”, nullable = false, length = 100) public Integer
getLeaves() {
return leaves;

                                                    }

public void setLeaves(Integer leaves) {
this.leaves = leaves;

                                                    }

@Column(name = “SALARY”, nullable = false, length = 100) public Integer
getSalary() {
return salary;

                                                    }

public void setSalary(Integer salary) {
this.salary = salary;

                                                    }

                                                    }
```

Lastly, we see the test for the ContractorEmployee entity which is similar to the test for the PermanentEmployee entity.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class ContractorEmployeeTest {
@Autowired
private SessionFactory sessionFactory;

@SuppressWarnings(“unchecked”)
@Test
public void testCRUD()
```

public void testCRUD()

{

```
Employee e1 = new Employee();
e1.setName(“A”);
sessionFactory.getCurrentSession().save(e1);


ContractorEmployee contractorEmployee = new ContractorEmployee();
contractorEmployee.setName(“Lalit Narayan Mishra”);
contractorEmployee.setHourlyRate(35);
contractorEmployee.setOvertimeRate(15);
sessionFactory.getCurrentSession().save(contractorEmployee);
contractorEmployee.setName(“Amritendu De”);
sessionFactory.getCurrentSession().merge(contractorEmployee);
List<ContractorEmployee> list =
sessionFactory.getCurrentSession().createQuery(“from
ContractorEmployee”).list(); Assert.assertEquals(1L, list.size());


sessionFactory.getCurrentSession().delete(contractorEmployee);
List<ContractorEmployee> list2 =
sessionFactory.getCurrentSession().createQuery(“from
ContractorEmployee”).list(); Assert.assertEquals(0L, list2.size());
```

}

}

Finally, we look at the ContractorEmployee entity definition.

```
@Entity
@Table(name=”CONTRACTOR_EMPLOYEE”)
@PrimaryKeyJoinColumn(name=”ID”)
public class ContractorEmployee extends Employee {


private Integer hourlyRate;
private Integer overtimeRate;
```

```
private Integer overtimeRate;


@Column(name = “HOURLEY_RATE”, nullable = false, length = 100) public
Integer getHourlyRate() {
return hourlyRate;

                                                    }

public void setHourlyRate(Integer hourlyRate) {
this.hourlyRate = hourlyRate;

                                                    }

@Column(name = “OVERTIME_RATE”, nullable = false, length = 100) public
Integer getOvertimeRate() {
return overtimeRate;

                                                    }

public void setOvertimeRate(Integer overtimeRate) {
this.overtimeRate = overtimeRate;

                                                    }

}
```

Table structure

```
DROP TABLE `permanent_employee`;
DROP TABLE `contractor_employee`;
DROP TABLE `employee`;
CREATE TABLE `employee` (
  `ID` int(11) NOT NULL,
  `NAME` varchar(100) NOT NULL,
  PRIMARY KEY (`ID`)

                                                    ) ;

CREATE TABLE `permanent_employee` (
```

```
  `LEAVES` int(11) NOT NULL,
  `SALARY` int(11) NOT NULL,
  `ID` int(11) NOT NULL,
  PRIMARY KEY (`ID`),
  CONSTRAINT FOREIGN KEY (`ID`) REFERENCES `employee` (`ID`) ) ;
CREATE TABLE `contractor_employee` (
  `HOURLEY_RATE` int(11) NOT NULL,
  `OVERTIME_RATE` int(11) NOT NULL,
  `ID` int(11) NOT NULL,
  PRIMARY KEY (`ID`),
  CONSTRAINT FOREIGN KEY (`ID`) REFERENCES `employee` (`ID`) ) ;
```

Develop the data access tier

We will begin with the test case for the find all records operation.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class EmployeeDaoImplTest {
@Autowired
private EmployeeDao dao ;

@Test
public void testGetAll() {
Assert.assertEquals(0L, dao.getAll().size());

                                                    }

                                                    }
```

We will then see how to write the corresponding data access operation of the above test.

```
@Repository
@Transactional
public class EmployeeDaoImpl  implements EmployeeDao {
@Autowired
private SessionFactory sessionFactory ;
```

```
@SuppressWarnings(“unchecked”)
@Override
public List<Employee> getAll() {
return sessionFactory.getCurrentSession().createQuery(“select employee from
Employee employee order by employee.id desc”).list(); }
```

}

Let’s move on to the insert operation test.

```
public class EmployeeDaoImplTest {
@Autowired
private EmployeeDao dao ;

@Test
public void testInsert() {
Employee temployee = new Employee();
temployee.setName(“Lalit Narayan Mishra”);
dao.insert(temployee);



Employee sEmployee = new Employee();
sEmployee.setName(“Amritendu De”);
dao.insert(sEmployee);



Assert.assertEquals(2L, dao.getAll().size());
}
```

}

The corresponding insert operation in the data access class has to be implemented using Hibernate 4 Session. save () function.

```
public class EmployeeDaoImpl  implements EmployeeDao {
@Autowired
private SessionFactory sessionFactory ;
@Override
public void insert(Employee employee) {
sessionFactory.getCurrentSession().save(employee);
}
```

}

We will then jump to the next test operation which is finding a row with an identifier.

```
public class EmployeeDaoImplTest {
@Test
public void testGetById() {
Employee semployee = new Employee();
semployee.setName("Lalit Narayan Mishra");
dao.insert(semployee);


List<Employee> pList = dao.getAll();
Employee employee = pList.get(0);


Employee employee2 = dao.getById(employee.getId());
Assert.assertEquals("Lalit Narayan Mishra", employee2.getName()); }
```

}

The corresponding data access method is executed using Hibernate 4 Session. get () operation.

```
public class EmployeeDaoImpl  implements EmployeeDao {
@Override
public Employee getById(Integer id) {
return (Employee) sessionFactory.getCurrentSession().get(Employee.class, id);
}
```

}

We will review the delete operation writing the test first, followed by the data access operation.

```
public class EmployeeDaoImplTest {
@Test
public void testDelete() {
Employee semployee = new Employee();
semployee.setName("Lalit Narayan Mishra");
dao.insert(semployee);
```

```
Employee tEmployee = new Employee();
tEmployee.setName(“Amritendu De”);
dao.insert(tEmployee);


Assert.assertEquals(2L, dao.getAll().size());


dao.delete(tEmployee);


Assert.assertEquals(1L, dao.getAll().size());
}
                                                    }
```

The delete data access operation uses Hibernate 4 Session.delete () operation.

```
public class EmployeeDaoImpl  implements EmployeeDao {
@Override
public void delete(Employee employee) {
sessionFactory.getCurrentSession().delete(employee); }
                                                    }
```

Last we will see the update test operation.

```
public class EmployeeDaoImplTest {
@Test
public void testUpdate() {
Employee semployee = new Employee();
semployee.setName(“Lalit Narayan Mishra”);
dao.insert(semployee);


Assert.assertEquals(1L, dao.getAll().size());

List<Employee> pList = dao.getAll();
Employee employee = pList.get(0);
```

```
Employee employee = pList.get(0);
employee.setName(“Amritendu De”);


dao.update(employee);


List<Employee> pList2 = dao.getAll();
Employee employee2 = pList2.get(0);
Assert.assertEquals(“Amritendu De”, employee2.getName()); }

                                                    }
```

The update data access operation is shown below which uses Hibernate 4 Session.merge () method.

```
public class EmployeeDaoImpl  implements EmployeeDao {
@Override
public void update(Employee employee) {
sessionFactory.getCurrentSession().merge(employee); }

                                                    }
```

Develop the business service tier

We will proceed with the test for the finding all records function.

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration( defaultRollback = true )
@Transactional
public class EmployeeServiceImplTest {
@Autowired
private EmployeeService service;
@Test
public void testFindAll() {
Assert.assertEquals(0L, service.findAll().size());
}

                                                    }
```

Let us look at how to code the corresponding find all records method in the business service.

```
@Service
@Transactional
public class EmployeeServiceImpl implements EmployeeService {

@Autowired
private EmployeeDao employeeDao;

@Autowired
private EmployeeMapper employeeMapper;

@Override
public List<EmployeeDto> findAll() {
List<Employee> employees = employeeDao.getAll(); List<EmployeeDto>
employeeDtos = new ArrayList<EmployeeDto>(); for(Employee employee :
employees){
employeeDtos.add(employeeMapper.mapEntityToDto(employee)); }
return employeeDtos;
}

                                                    }
```

The mapper characteristically has two vital operations which are used to convert data access objects to entities and vice versa.

```
@Component
public class EmployeeMapper {

public Employee mapDtoToEntity(EmployeeDto employeeDto){
if(employeeDto instanceof PermanentEmployeeDto) {
PermanentEmployee permanentEmployee = new PermanentEmployee();
if(null!=employeeDto.getId()) permanentEmployee.setId(employeeDto.getId());
if(null!=employeeDto.getName())
permanentEmployee.setName(employeeDto.getName()); if(null!=
((PermanentEmployeeDto)employeeDto).getLeaves())
permanentEmployee.setLeaves(((PermanentEmployeeDto)employeeDto).getLeav
if(null!=((PermanentEmployeeDto)employeeDto).getSalary())
permanentEmployee.setSalary(((PermanentEmployeeDto)employeeDto).getSalar
return permanentEmployee;
} else if(employeeDto instanceof ContractorEmployeeDto) {
```

```
} else if(employeeDto instanceof ContractorEmployeeDto) {
ContractorEmployee contractorEmployee = new ContractorEmployee();
if(null!=employeeDto.getId()) contractorEmployee.setId(employeeDto.getId());
if(null!=employeeDto.getName())
contractorEmployee.setName(employeeDto.getName()); if(null!=
((ContractorEmployeeDto)employeeDto).getHourlyRate())
contractorEmployee.setHourlyRate(((ContractorEmployeeDto)employeeDto).get
if(null!=((ContractorEmployeeDto)employeeDto).getOvertimeRate())
contractorEmployee.setOvertimeRate(((ContractorEmployeeDto)employeeDto).g
return contractorEmployee;

                                    }

return null;

                                    }

public EmployeeDto mapEntityToDto(Employee employee){
if(employee instanceof PermanentEmployee) {
PermanentEmployeeDto permanentEmployeeDto = new
PermanentEmployeeDto(); if(null!=employee.getId())
permanentEmployeeDto.setId(employee.getId()); if(null!=employee.getName())
permanentEmployeeDto.setName(employee.getName()); if(null!=
((PermanentEmployee)employee).getLeaves())
permanentEmployeeDto.setLeaves(((PermanentEmployee)employee).getLeaves(
if(null!=((PermanentEmployee)employee).getSalary())
permanentEmployeeDto.setSalary(((PermanentEmployee)employee).getSalary())
return permanentEmployeeDto;
} else if(employee instanceof ContractorEmployee) {
ContractorEmployeeDto contractorEmployeeDto = new
ContractorEmployeeDto(); if(null!=employee.getId())
contractorEmployeeDto.setId(employee.getId()); if(null!=employee.getName())
contractorEmployeeDto.setName(employee.getName()); if(null!=
((ContractorEmployee)employee).getHourlyRate())
contractorEmployeeDto.setHourlyRate(((ContractorEmployee)employee).getHou
if(null!=((ContractorEmployee)employee).getOvertimeRate())
contractorEmployeeDto.setOvertimeRate(((ContractorEmployee)employee).getO
return contractorEmployeeDto;

                                    }
```

```
return null;

                                                    }

                                                    }
```

The following create test operation is used to create a new row in the database.

```
public class EmployeeServiceImplTest {

@Test
public void testCreate() {
EmployeeDto bemployee = new PermanentEmployeeDto();
bemployee.setName(“Lalit Narayan Mishra”);
((PermanentEmployeeDto)bemployee).setLeaves(30);
((PermanentEmployeeDto)bemployee).setSalary(500000);
service.create(bemployee);


EmployeeDto hEmployee = new ContractorEmployeeDto();
hEmployee.setName(“Amritendu De”);
((ContractorEmployeeDto)hEmployee).setHourlyRate(35);
((ContractorEmployeeDto)hEmployee).setOvertimeRate(20);
service.create(hEmployee);


Assert.assertEquals(2L, service.findAll().size());
}

                                                    }
```

The matching method in the business service tier accepts a data access object, converts to entity and delegates to the data access tier to commit to the database.

```
public class EmployeeServiceImpl implements EmployeeService {

@Override
public void create(EmployeeDto employeeDto) {
employeeDao.insert(employeeMapper.mapDtoToEntity(employeeDto)); }

                                                    }
```

The next operation we will define is finding a row with a given identifier.

```
public class EmployeeServiceImplTest {
@Test
public void testFindById() {
EmployeeDto bemployee = new PermanentEmployeeDto();
bemployee.setName(“Lalit Narayan Mishra”);
((PermanentEmployeeDto)bemployee).setLeaves(30);
((PermanentEmployeeDto)bemployee).setSalary(500000);
service.create(bemployee);


List<EmployeeDto> pList = service.findAll();
EmployeeDto employee = pList.get(0);


EmployeeDto employee2 = service.findById(employee.getId());
Assert.assertEquals(“Lalit Narayan Mishra”, employee2.getName()); }

                                                    }
```

The equivalent operation of the business service is to find a row with a given identifier.

```
public class EmployeeServiceImpl implements EmployeeService {
@Override
public EmployeeDto findById(Integer id) {
Employee employee = employeeDao.getById(id);
EmployeeDto employeeDto = null;
if(null !=employee){
employeeDto = employeeMapper.mapEntityToDto(employee); }
return employeeDto;
}

                                                    }
```

The remove procedure is then in line which accepts a given identifier and removes the alike row from the database.

```
public class EmployeeServiceImplTest {
@Test
public void testRemove() {
```

```
EmployeeDto bemployee = new PermanentEmployeeDto();
bemployee.setName(“Lalit Narayan Mishra”);
((PermanentEmployeeDto)bemployee).setLeaves(30);
((PermanentEmployeeDto)bemployee).setSalary(500000);
service.create(bemployee);


EmployeeDto hEmployee = new ContractorEmployeeDto();
hEmployee.setName(“Amritendu De”);
((ContractorEmployeeDto)hEmployee).setHourlyRate(35);
((ContractorEmployeeDto)hEmployee).setOvertimeRate(20);
service.create(hEmployee);


Assert.assertEquals(2L, service.findAll().size());


List<EmployeeDto> pList = service.findAll();
EmployeeDto employee = pList.get(0);
service.remove(employee.getId());


Assert.assertEquals(1L, service.findAll().size());

                                                    }

                                                    }
```

The equivalent business service method is to be demarcated next which accepts an identifier and removes the matching instance from the database via the data access tier operation.

```
public class EmployeeServiceImpl implements EmployeeService {
@Override
public void remove(Integer employeeId) {
Employee employee = employeeDao.getById(employeeId);
employeeDao.delete(employee);
}
}
```

The last test operation which we will do is the edit operation which takes a data access object and applies the data access method to merge the changes into the database.

```
public class EmployeeServiceImplTest {
@Test
public void testEdit() {
EmployeeDto bemployee = new PermanentEmployeeDto();
bemployee.setName(“Lalit Narayan Mishra”);
((PermanentEmployeeDto)bemployee).setLeaves(30);
((PermanentEmployeeDto)bemployee).setSalary(500000);
service.create(bemployee);


Assert.assertEquals(1L, service.findAll().size());

List<EmployeeDto> list = service. findAll (); EmployeeDto employee =
pList.get(0);
employee.setName(“Amritendu De”);


service.edit(employee);


List<EmployeeDto> pList2 = service.findAll(); EmployeeDto employee2 =
pList2.get(0);
Assert.assertEquals(“Amritendu De”, employee2.getName()); }

                                                    }
```

The corresponding operation of the business service is the edit operation which is defined below.

```
public class EmployeeServiceImpl implements EmployeeService {
 @Override
public void edit(EmployeeDto employeeDto) {
employeeDao.update(employeeMapper.mapDtoToEntity(employeeDto)); }

                                                    }
```

Develop the presentation tier

The last level in the stack is the REST based Spring Controller. First, we will look at how to write the test and then follow with the corresponding controller.

```
@RunWith(SpringJUnit4ClassRunner.class)
@WebAppConfiguration
@ContextConfiguration( locations = { “classpath:context.xml” } )
@TransactionConfiguration(defaultRollback = true)
@Transactional
public class EmployeeControllerTest {

private Gson gson = new GsonBuilder().create();

@Resource
private WebApplicationContext webApplicationContext;
private MockMvc mockMvc;

@Before
public void setUp() {
mockMvc = MockMvcBuilders.<StandaloneMockMvcBuilder>
webAppContextSetup(webApplicationContext).build(); }

@Test
public void testAll() throws Exception {
testCreate();
testUpdate();
testDelete();

                                        }


public void testCreate() throws Exception {


EmployeeDto employeeDto = new PermanentEmployeeDto();
employeeDto.setName(“ABC”);
((PermanentEmployeeDto)employeeDto).setLeaves(20);
((PermanentEmployeeDto)employeeDto).setSalary(1000000); String json =
gson.toJson(employeeDto);
```

```
gson.toJson(employeeDto);
json= json.replace("{", "");
json= json.replace("}", "");
json = "{\"type\":\"permanentEmployee\","+json+"}";
MockHttpServletRequestBuilder requestBuilderOne =
MockMvcRequestBuilders.post("classtableinheritancecreate");
requestBuilderOne.contentType(MediaType.APPLICATION_JSON);
requestBuilderOne.content(json.getBytes());
this.mockMvc.perform(requestBuilderOne).andExpect(MockMvcResultMatchers
}


public void testUpdate() throws Exception {


MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("classtableinheritancefindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString();
Type listType = new TypeToken<EmployeeDto[]>() {}.getType();
EmployeeDto[] employeeDtoList = gson.fromJson(response2, listType);
EmployeeDto employeeDto2 = employeeDtoList[0];


EmployeeDto newEmployeeDto = new PermanentEmployeeDto();
newEmployeeDto.setId(employeeDto2.getId());
newEmployeeDto.setName("DEF");
((PermanentEmployeeDto)newEmployeeDto).setLeaves(20);
((PermanentEmployeeDto)newEmployeeDto).setSalary(1000000);
String json2 = gson.toJson(newEmployeeDto);
json2= json2.replace("{", "");
json2= json2.replace("}", "");
json2 = "{\"type\":\"permanentEmployee\","+json2+"}";
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("classtableinheritanceedit");
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
requestBuilder3.content(json2.getBytes());
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
```

```
}

public void testDelete() throws Exception {


MockHttpServletRequestBuilder requestBuilder2 =
MockMvcRequestBuilders.get("classstableinheritancefindAll"); MvcResult
result = this.mockMvc.perform(requestBuilder2).andReturn(); String response2
= result.getResponse().getContentAsString();
Type listType = new TypeToken<EmployeeDto[]>() {}.getType();
EmployeeDto[] employeeDtoList = gson.fromJson(response2, listType);
EmployeeDto employeeDto2 = employeeDtoList[0];
MockHttpServletRequestBuilder requestBuilder3 =
MockMvcRequestBuilders.post("classstableinheritanceremove/"+employeeDto2.
requestBuilder3.contentType(MediaType.APPLICATION_JSON);
this.mockMvc.perform(requestBuilder3).andExpect(MockMvcResultMatchers.st
}

                                    }
```

The matching REST, Spring Controller, will be contacting the business service tier to connect to the database. Please note that the code is like the mock controller developed earlier, with the difference being in the request mapping and the call going to the actual database instead of the mock in-memory database.

```
@Controller
@RequestMapping(value="/classtableinheritance")
@Transactional
public class EmployeeController {

@Autowired
private EmployeeService service;


@RequestMapping(value="/findAll", method=RequestMethod.GET) public
@ResponseBody EmployeeDto[] findAll(){
List<EmployeeDto> list = service.findAll();
return list.toArray(new EmployeeDto[list.size()]);
```

```
}

@RequestMapping(value="/findById/{employeeid}",
method=RequestMethod.GET) public @ResponseBody EmployeeDto
findById(@PathVariable("employeeid") Integer employeeid){
return service.findById(employeeid);

}

@RequestMapping(value="/create", method=RequestMethod.POST)
@ResponseBody
public void create(@RequestBody EmployeeDto employee){
service.create(employee);

}

@RequestMapping(value="/remove/{employeeid}",
method=RequestMethod.POST) @ResponseStatus(value =
HttpStatus.NO_CONTENT)
public void remove(@PathVariable("employeeid") Integer employeeid){
service.remove(employeeid);

}

@RequestMapping(value="/edit", method=RequestMethod.POST)
@ResponseBody
public void edit(@RequestBody EmployeeDto employee){
service.edit(employee);

}

}
```

We have reached the end of the chapter where we will discuss how to integrate the development work done by both of the teams. The REST service will need to be integrated with the actual user interface. Please note that the request mapping changes from "*classtetableinheritance*mock" to "/classtableinheritance". The remaining work involves creating the actual user interface by taking most of the contents of the mock user interface.

Summary

In this chapter we discussed the following:

- Develop mock in-memory database
- Develop mock user interface
- Develop entity related to class table inheritance
- Develop data access tier
- Develop business service tier
- Develop a REST service controller.

Appendix: Setting Up the Workspace

This chapter briefs you on how to work with the GitHub source control location and configure the workspace using Eclipse. Following the chapter closely will enable you to configure the application and run it successfully. The software versions are as of this writing. At a later date, you may choose to download the latest versions, but the applications may require a certain amount of tweaking to run perfectly.

Steps to Set Up Spring OODD Workspace

Prerequisites for running Spring OODD application
Eclipse IDE for Java EE Developers (http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/heliossr2) **JDK 1.7** (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html) **Apache Maven** (http://maven.apache.org/) **MySql DB Server** (http://www.mysql.com/) **Apache Tomcat** (http://tomcat.apache.org/)

Download the source from GitHub

The source can be downloaded from GitHub (https://github.com/Spring-Hibernate-Book/spring-hibernate-datamodeling-tdd-rest) as shown in Figure 17-1.

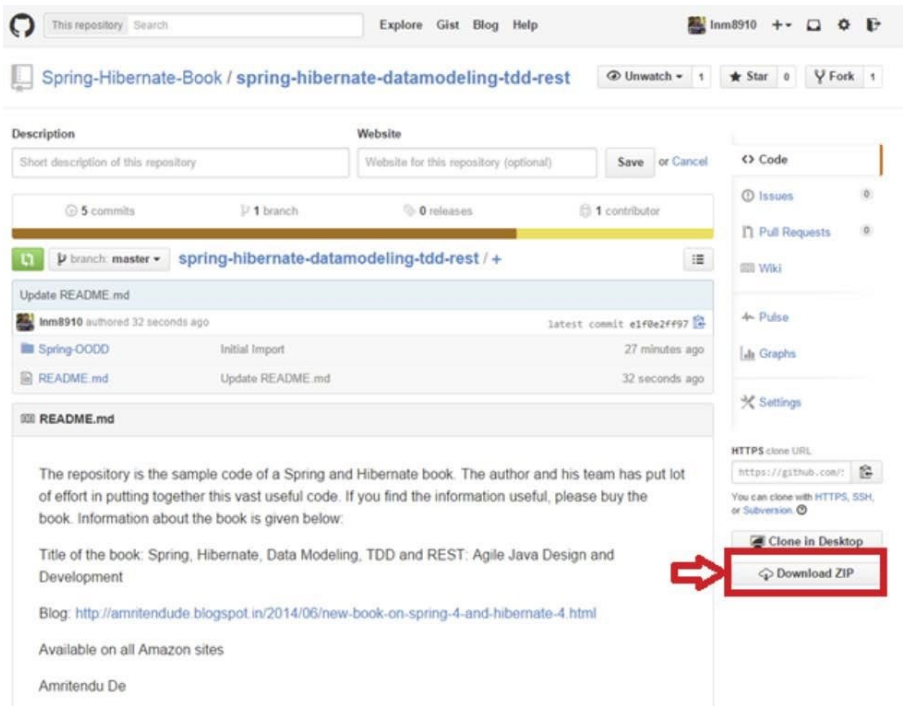

Figure 17-1: Download the code from GitHub

Unzip the downloaded file and import into Eclipse as Maven Project The file downloaded from GitHub should be unzipped and fed as an input to Eclipse by means of a Maven project as shown in Figure 17-2.

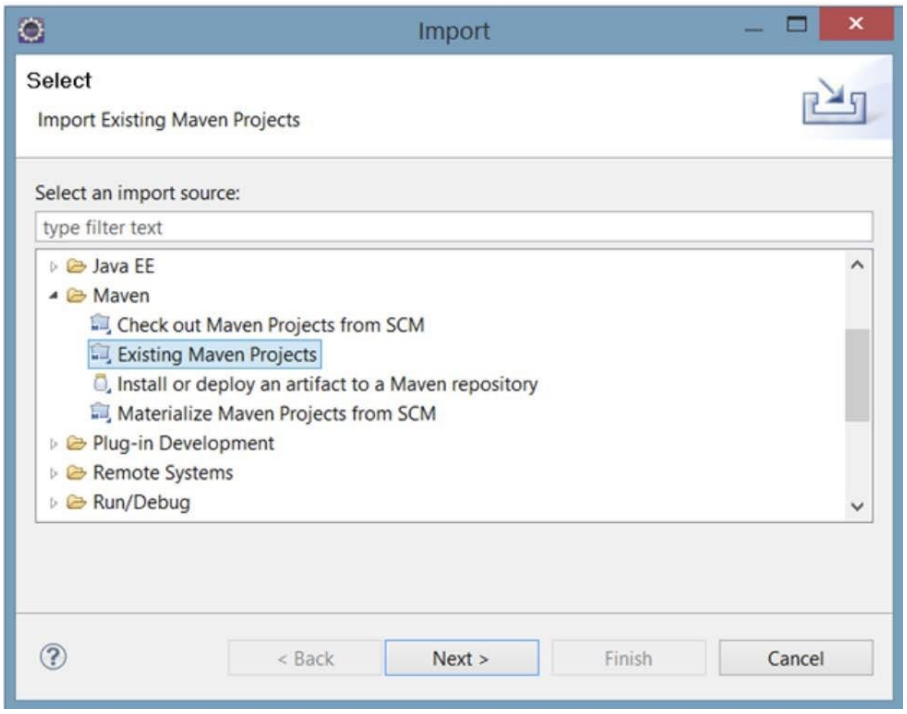

Figure 17-2: Import into Eclipse as Maven project

After choosing Existing Maven Projects, select the folder where you have unzipped the downloaded file and choose the directory containing the POM as shown in Figure 17-3.

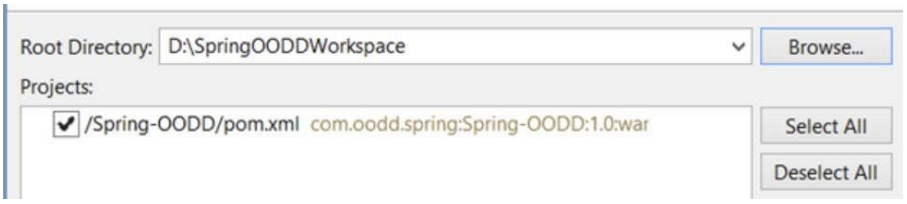

Figure 17-3: Select the POM inside the unzipped folder

Expanded view of project in Eclipse

After the project has been imported, a view of the project structure is shown in Figure 17-4.

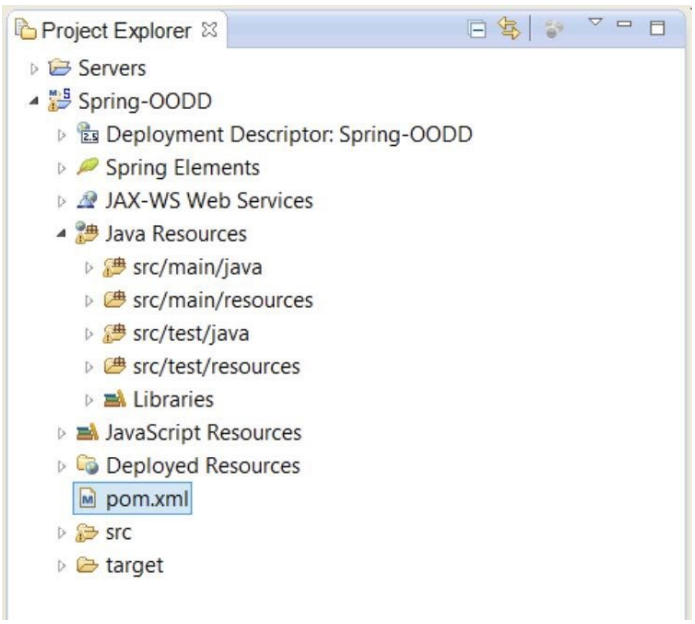

Figure 17-4: A view of the project structure

Configure database schema

After you have installed MySQL, create a new schema named **spring** using MySql Workbench, as shown in Figure 17-5. Please note that the database is agnostic and any other database can be used in place of MySQL. But the code has been tested only on MySQL so other databases may require small tweaks.

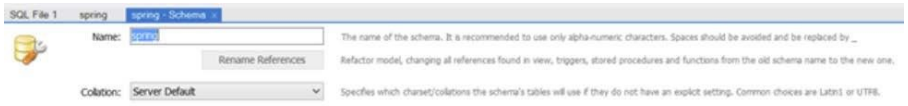

Figure 17-5: Create schema using MySQL Workbench

Verify & add the credentials in datasource.properties file

The credentials are pre-configured in datasource.properties file having two instances – one for main and the other for test. You can verify them as shown in Figure 17-6.

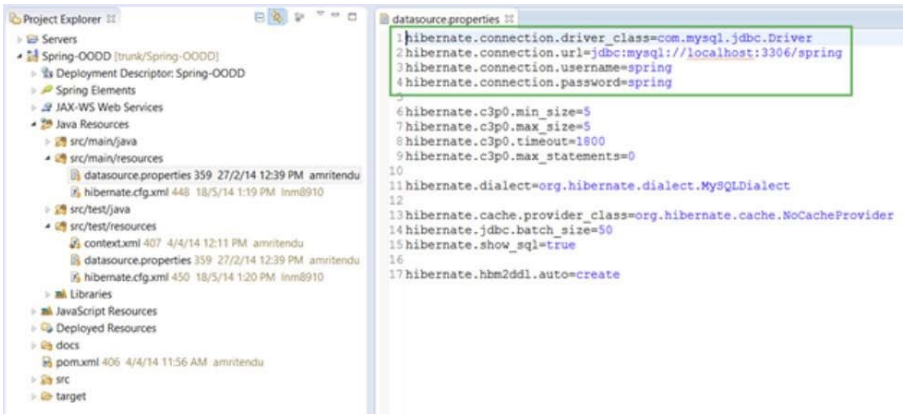

Figure 17-6: Verify the credentials in datasource.properties

Add Tomcat Server in Eclipse and add Spring-OODD Project to Tomcat The application has been tested using Tomcat Server. You are free to use any other JEE Application Server to deploy the application. Choose appropriate Tomcat Installation as shown in Figure 17-7.

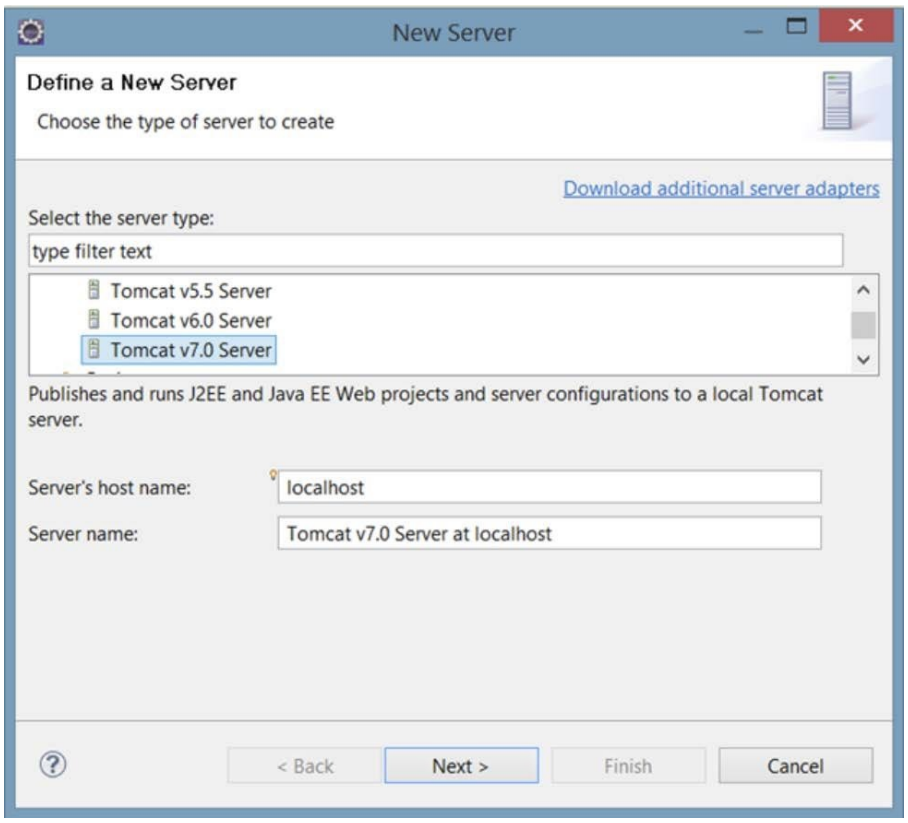

Figure 17-7: Choose the version of Tomcat Server

As of this writing, we have used Java SE 7 to test the application which is the most recent version released by Oracle. You may configure the application with a higher version of Java SE and Tomcat installation. There may be some small tweaks required as applicable. Right click on Project and go to Properties as shown in Figure 17-8.

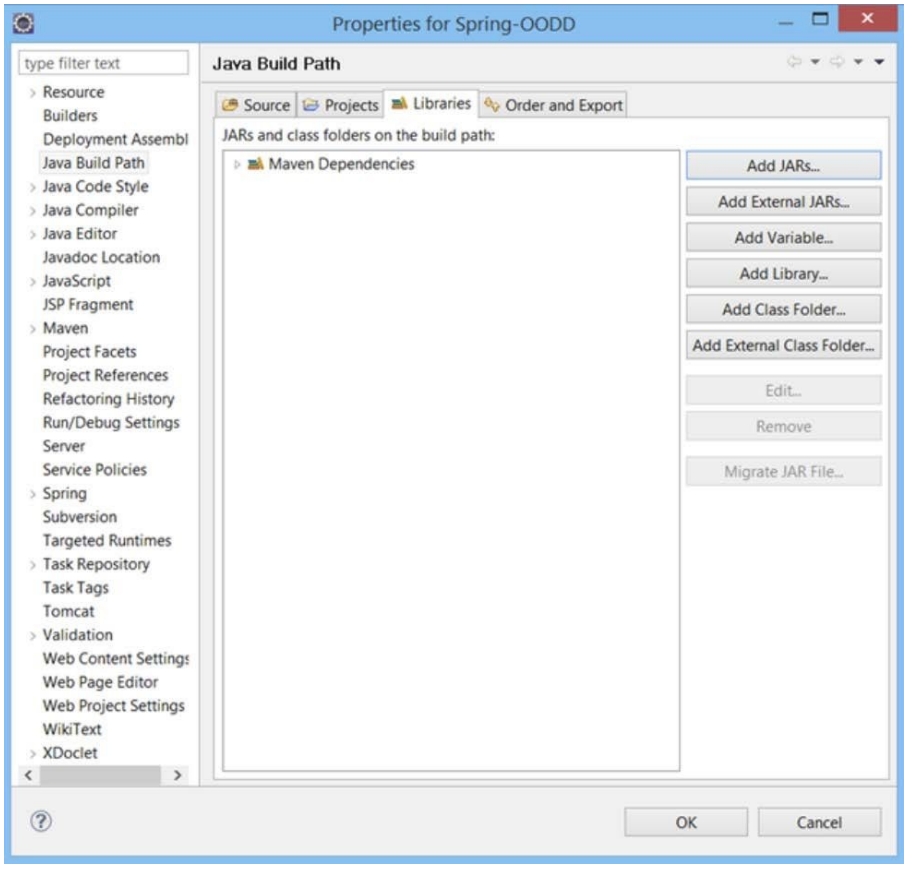

Figure 17-8: Properties window of Project

Click on Add Library ◊JRE System Library ◊ Alternate JRE ◊ Installed JREs. Here choose your JDK Installation as shown in Figure 17-9.

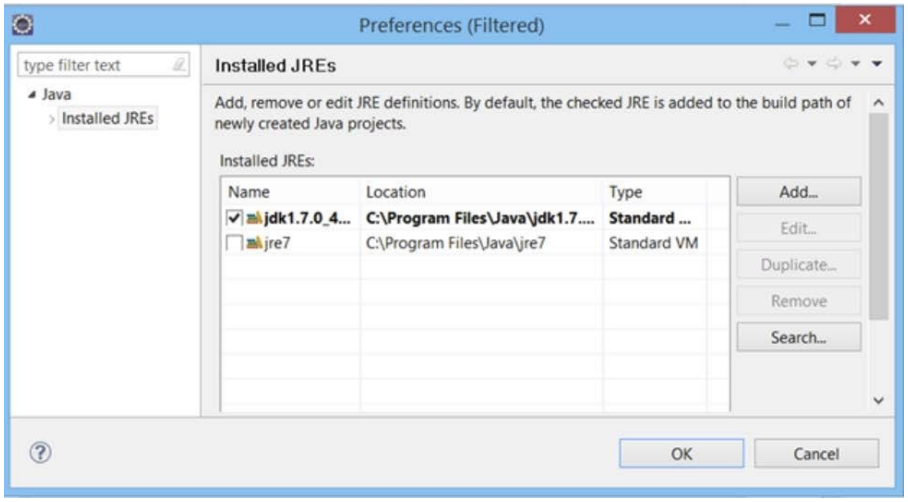

Figure 17-9: Installed JREs Window

Set it as Workspace Default and click on Finish, as shown in Figure 17-10.

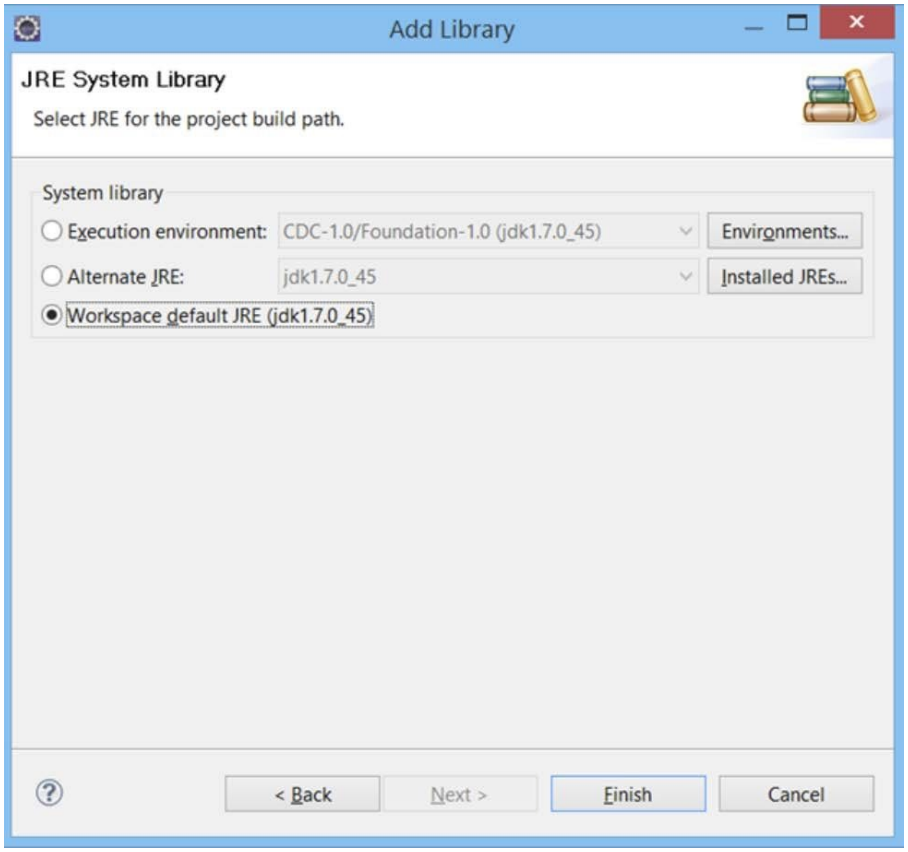

Figure 17-10: JRE System Library Window

Choose appropriate JRE in Tomcat Server window, as shown in Figure 17-11.

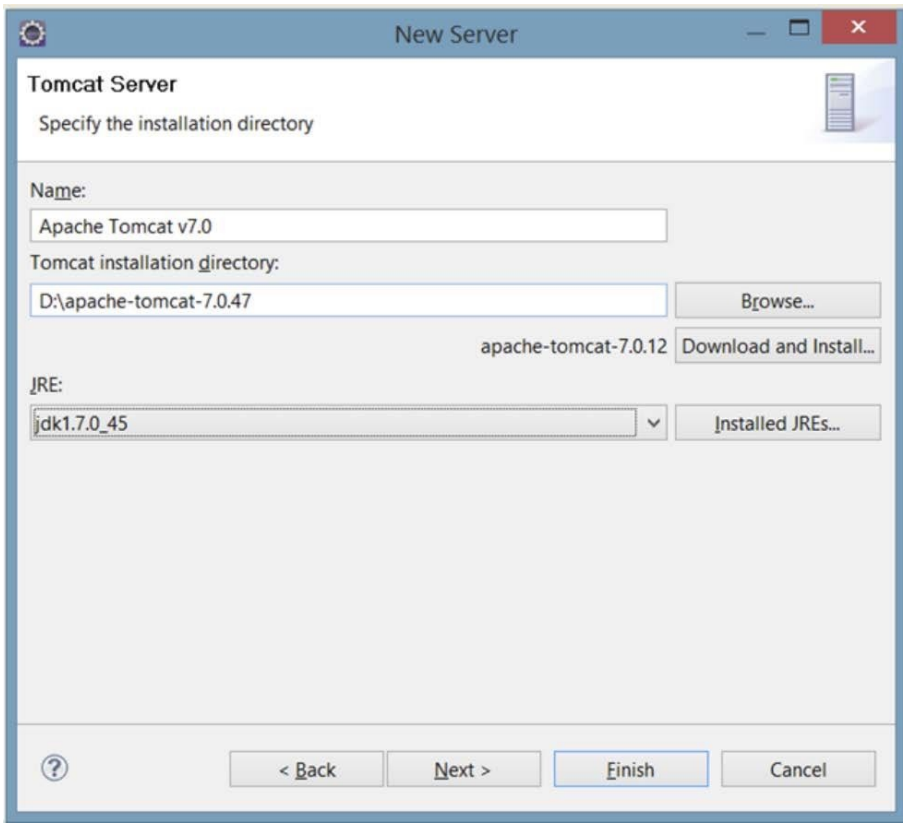

Figure 17-11: Choose the JDK version configured

Add Spring-OODD Project to the server

The Spring-OODD project should be added to the Tomcat configuration to be deployed, as shown in Figure 17-12.

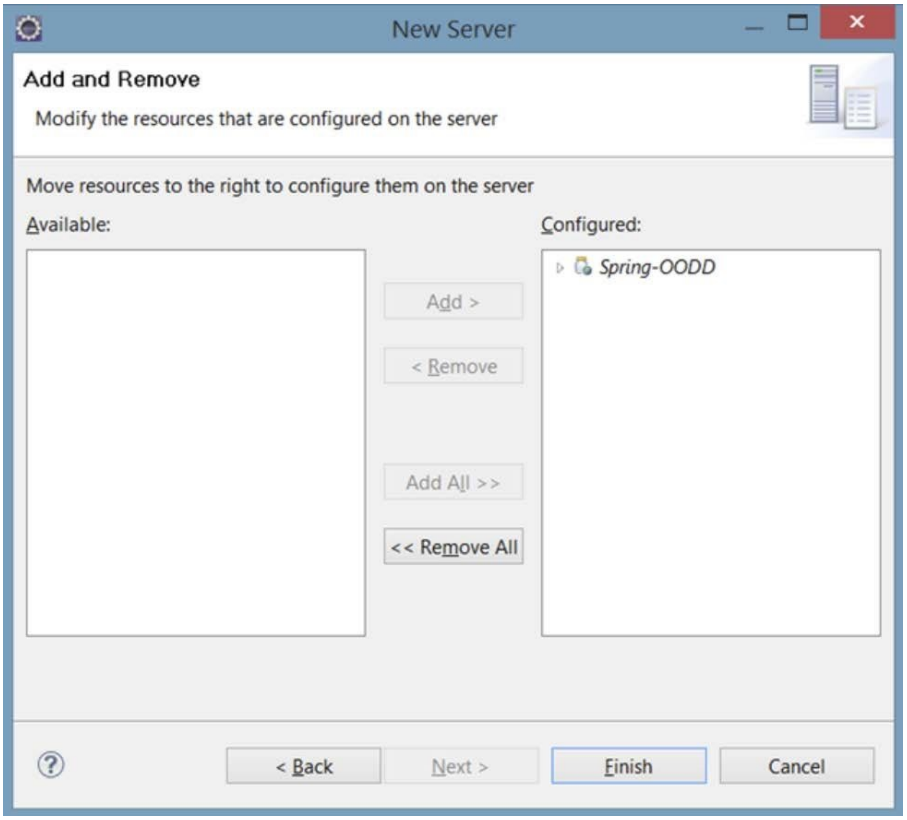

Figure 17-12: Choose the Spring-OODD Project for deployment to Tomcat Server **Start Tomcat and run the application**

After Tomcat starts successfully the index page will be loaded automatically as configured in web.xml. All the modules of the application can be tested using the index page as shown in Figure 17-13. The default url is http://localhost:8080/Spring-OODD/

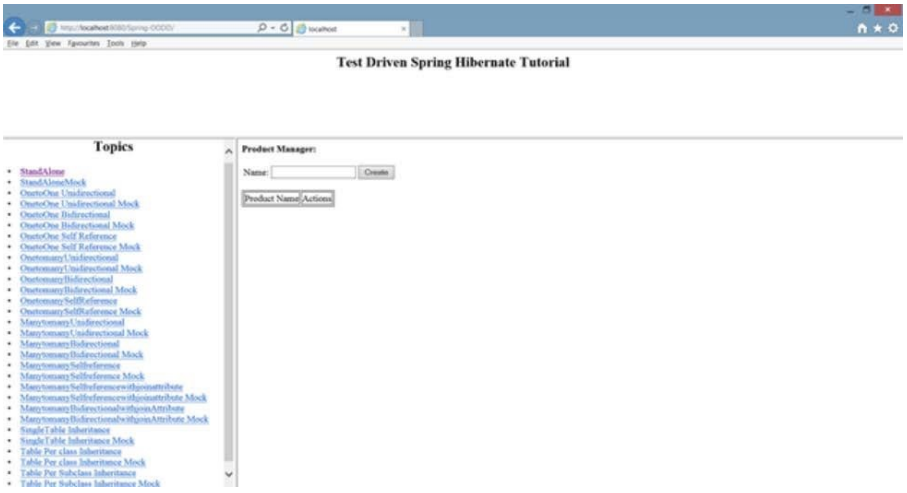

Figure 17-13: The main index page of the application

Summary

As a last note, I want to let you know that this application has been developed with a lot of hard work and diligence. Please do not share the application on groups as a free resource. However, you are welcome to use the concepts in a day-to-day job.