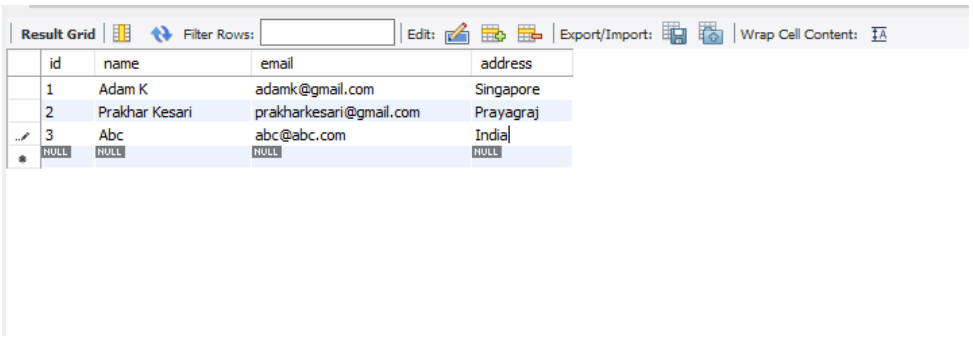
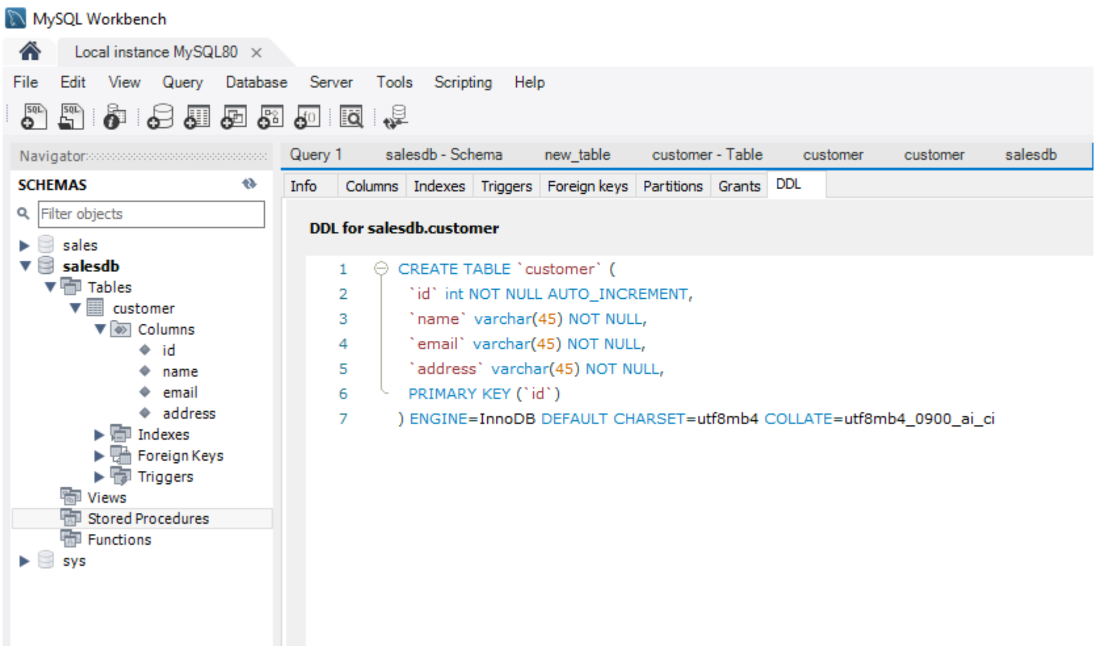
Customer Account Tracker – Case Study

The data is stored in a database to maintain the record of students.



id	name	email	address
1	Adam K	adamk@gmail.com	Singapore
2	Prakhar Kesari	prakharkesari@gmail.com	Prayagraj
3	Abc	abc@abc.com	India
NULL	NULL	NULL	NULL

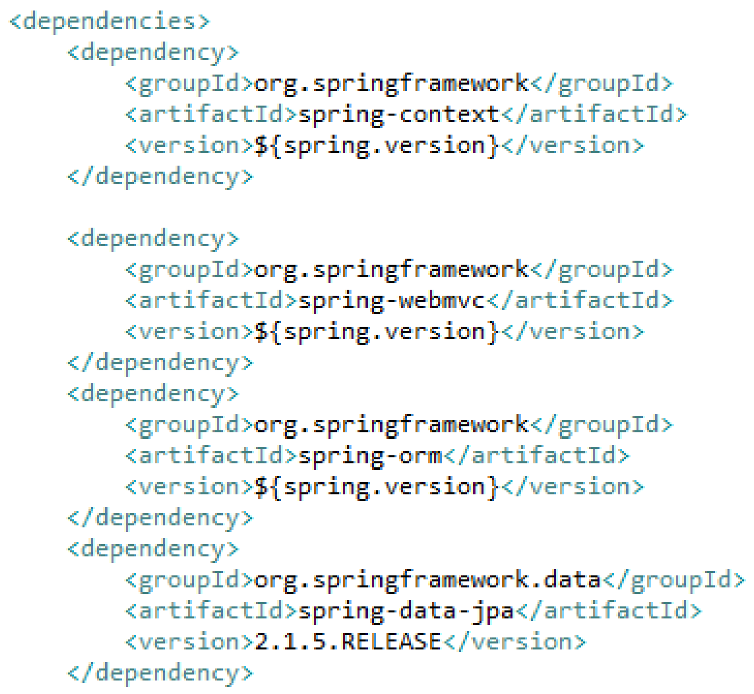
Database is created using mySQL queries and the values are stored



DDL for salesdb.customer

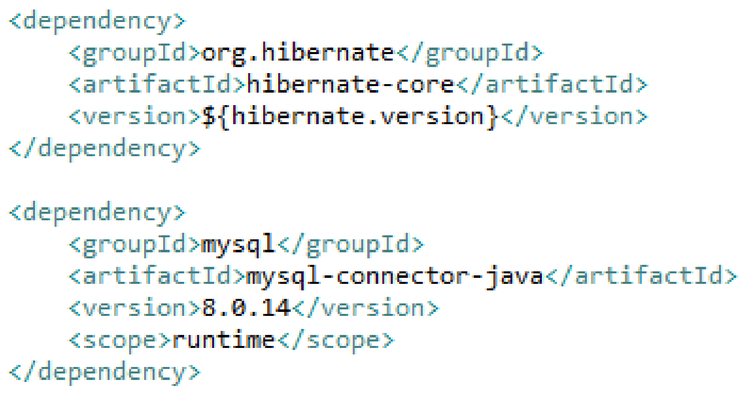
```sql
CREATE TABLE `customer` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(45) NOT NULL,
  `email` varchar(45) NOT NULL,
  `address` varchar(45) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Various Spring dependencies are injected to perform spring functions in pom.xml



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

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>2.1.5.RELEASE</version>
    </dependency>
```

Hibernate dependencies are added to interact with mySQL database



```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>${hibernate.version}</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.14</version>
    <scope>runtime</scope>
</dependency>
```

Dependencies to add JSP, Servlets and Jstl

```
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.1.0</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>javax.servlet.jsp-api</artifactId>
        <version>2.3.1</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>
```

Code Model Class

Create the domain class **Customer** to map with the table customer in the database as following:

```java
package com.wipro.customer;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
    private String address;

    protected Customer() {
    }

    protected Customer(String name, String email, String address) {
        this.name = name;
        this.email = email;
        this.address = address;
    }

```

the annotation `@Entity` to map this class to the table customer (the class has same name as the table). All the class' field names are also identical to the table's ones. The field `id` is annotated with `@Id` and `@GeneratedValue` annotations to indicate that this field is primary key and its value is auto generated.

Next, Java code to configure Spring MVC and Spring Data JPA. Java-based configuration are used as it's simpler than XML.

```java
package com.wipro.config;

import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;

import org.springframework.web.WebApplicationInitializer;
import org.springframework.web.context.support.AnnotationConfigWebApplicationContext;
import org.springframework.web.servlet.DispatcherServlet;

public class WebAppInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext appContext = new AnnotationConfigWebApplicationContext();
        appContext.register(WebMvcConfig.class);

        ServletRegistration.Dynamic dispatcher = servletContext.addServlet(
                "SpringDispatcher", new DispatcherServlet(appContext));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/");

    }
}
```

The `onStartup()` method of this class will be automatically invoked by the servlet container when the application is being loaded. The Spring Dispatcher Servlet handles all the requests via the URL mapping "/" and it looks for configuration in the `WebMvcConfig` class, which is described below.

Configure Spring MVC:

Creating the **`WebMvcConfig`** class under the `com.wipro.config` package with the following code:

```java
package com.wipro.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

@Configuration
@ComponentScan("net.codejava ")
public class WebMvcConfig {
    @Bean(name = "viewResolver")
    public InternalResourceViewResolver getViewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/views/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }
}
```

This class is annotated with the `@Configuration` annotation to tell Spring framework that this is a configuration class. The `@ComponentScan` annotation tells Spring to scan for configuration classes in the `com.wipro` package.

In this class, we simply create a view resolver bean that specifies the prefix and suffix for view files. So create the directory `views` under `WebContent/WEB-INF` directory to store JSP files.

Configure Spring Data JPA:

To enable Spring Data JPA, we need to create two beans: `EntityManagerFactory` and `JpaTransactionManager`. So create another configuration class named **`JpaConfig`** with the following code:

```java
package com.wipro.config;

import javax.persistence.EntityManagerFactory;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalEntityManagerFactoryBean;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableJpaRepositories(basePackages = {"net.codejava.customer"})
@EnableTransactionManagement
public class JpaConfig {
    @Bean
    public LocalEntityManagerFactoryBean entityManagerFactory() {
        LocalEntityManagerFactoryBean factoryBean = new LocalEntityManagerFactoryBean();
        factoryBean.setPersistenceUnitName("SalesDB");

        return factoryBean;
    }

    @Bean
    public JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
        JpaTransactionManager transactionManager = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(entityManagerFactory);

        return transactionManager;
    }
}
```

Here, two important annotations are used:

- **`@EnableJpaRepositories`**: this tells Spring Data JPA to look for repository classes in the specified package (com.wipro) in order to inject relevant code at runtime.

- **`@EnableTransactionManagement`**: this tells Spring Data JPA to generate code for transaction management at runtime.

In this class, the first method creates an instance of `EntityManagerFactory` to manage the persistence unit `SalesDB` (this name is specified in the `persistence.xml` file above).

And the last method creates an instance of `JpaTransactionManager` for the `EntityManagerFactory` created by the first method.

That's the minimum required configuration for using Spring Data JPA.

Code Repository Interface

Next, create the `CustomerRepository` interface that extends the `CrudRepository` interface defined by Spring Data JPA with the following code:

```java
package com.wipro.customer;

import java.util.List;

import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

public interface CustomerRepository extends CrudRepository<Customer, Long> {

}
```

You see, this is almost the code we need for the data access layer. Deadly simple, right? As with Spring Data JPA, you don't have to write any DAO code. Just declare an interface that extends the `CrudRepository` interface, which defines CRUD methods like `save()`, `findAll()`, `findById()`, `deleteById()`, etc. At runtime, Spring Data JPA automatically generates the implementation code.

Note that we must specify the type of the model class and type of the primary key field when extending the `CrudRepository` interface: `CrudRepository<Customer, Long>`

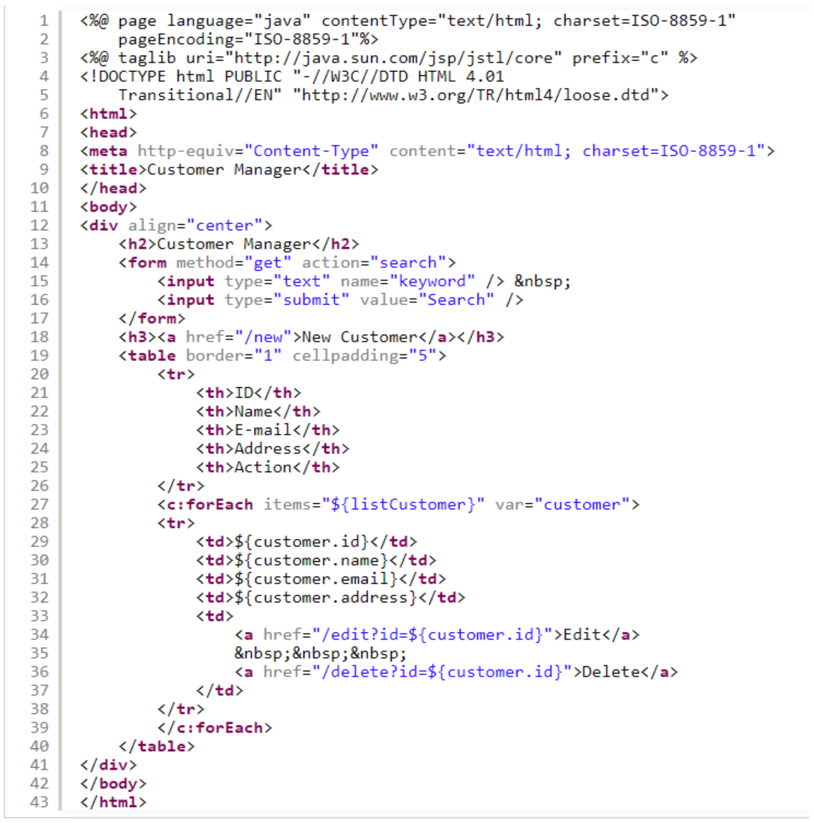
Code Spring MVC Controller Class

Next, in the controller layer, create the `CustomerController` class to handle all requests from the clients with the following code:

```java
package com.wipro.customer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;


@Controller
public class CustomerController {

    @Autowired
    private CustomerService customerService;

    // handler methods will go here...
}
```

This is a typical Spring MVC controller class, which is annotated with the `@Controller` annotation. You can see an instance of `CustomerService` is injected into this class using the `@Autowired` annotation.

Write code for the handler methods in the following sections.



```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01
    Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Customer Manager</title>
</head>
<body>
<div align="center">
    <h2>Customer Manager</h2>
    <form method="get" action="search">
        <input type="text" name="keyword" /> &nbsp;
        <input type="submit" value="Search" />
    </form>
    <h3><a href="/new">New Customer</a></h3>
    <table border="1" cellpadding="5">
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>E-mail</th>
            <th>Address</th>
            <th>Action</th>
        </tr>
        <c:forEach items="${listCustomer}" var="customer">
        <tr>
            <td>${customer.id}</td>
            <td>${customer.name}</td>
            <td>${customer.email}</td>
            <td>${customer.address}</td>
            <td>
                <a href="/edit?id=${customer.id}">Edit</a>
                &nbsp;&nbsp;&nbsp;
                <a href="/delete?id=${customer.id}">Delete</a>
            </td>
        </tr>
        </c:forEach>
    </table>
</div>
</body>
</html>
```

Run the website application. Add some rows in the table customer and access the URL *http://localhost:8080/CustomerManager/*, one should see something like this:

Code Create New Customer Feature

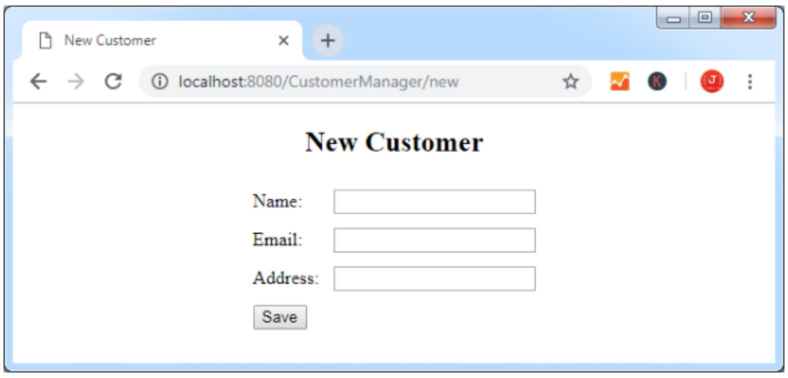
To implement the create new customer feature, we need to write two handler methods. The first one is to display the new customer form:

```java
@RequestMapping("/new")
public String newCustomerForm(Map<String, Object> model) {
    Customer customer = new Customer();
    model.put("customer", customer);
    return "new_customer";
}
```

And write code for the JSP page `new_customer.jsp` as follows:

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>New Customer</title>
</head>
<body>
    <div align="center">
        <h2>New Customer</h2>
        <form:form action="save" method="post" modelAttribute="customer">
            <table border="0" cellpadding="5">
                <tr>
                    <td>Name: </td>
                    <td><form:input path="name" /></td>
                </tr>
                <tr>
                    <td>Email: </td>
                    <td><form:input path="email" /></td>
                </tr>
                <tr>
                    <td>Address: </td>
                    <td><form:input path="address" /></td>
                </tr>
                <tr>
                    <td colspan="2"><input type="submit" value="Save"></td>
                </tr>
            </table>
        </form:form>
    </div>
</body>
</html>
```

Click the link **New Customer** in the home page, the new customer form looks like this:



New Customer

Name:	
Email:	
Address:	
Save	

And the second handler method is to handle the Save button on this form:

```java
@RequestMapping(value = "/save", method = RequestMethod.POST)
public String saveCustomer(@ModelAttribute("customer") Customer customer) {
    customerService.save(customer);
    return "redirect:/";
}
```

As you can see, it will redirect the client to the home page after the customer has been saved successfully.

Code Edit Customer Feature

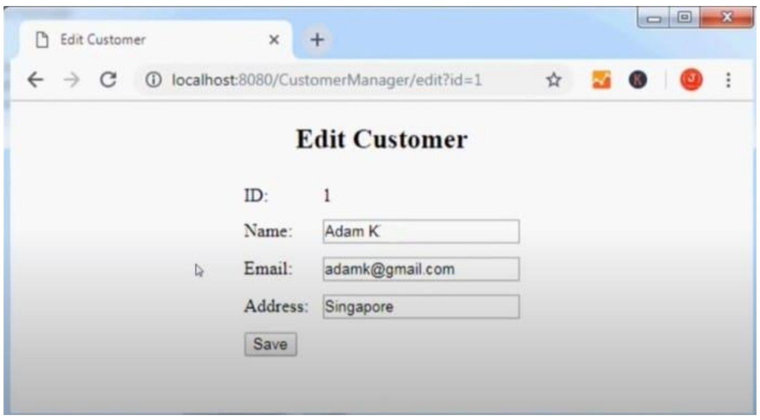
To implement the edit/update customer feature, add the following handler method to the `CustomerController` class:

```java
@RequestMapping("/edit")
public ModelAndView editCustomerForm(@RequestParam long id) {
    ModelAndView mav = new ModelAndView("edit_customer");
    Customer customer = customerService.get(id);
    mav.addObject("customer", customer);

    return mav;
}
```

This method will show the Edit customer form, so code the `edit_customer.jsp` file as follows:

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Edit Customer</title>
</head>
<body>
    <div align="center">
        <h2>Edit Customer</h2>
        <form:form action="save" method="post" modelAttribute="customer">
            <table border="0" cellpadding="5">
                <tr>
                    <td>ID: </td>
                    <td>${customer.id}
                        <form:hidden path="id"/>
                    </td>
                </tr>
                <tr>
                    <td>Name: </td>
                    <td><form:input path="name" /></td>
                </tr>
                <tr>
                    <td>Email: </td>
                    <td><form:input path="email" /></td>
                </tr>
                <tr>
                    <td>Address: </td>
                    <td><form:input path="address" /></td>
                </tr>
                <tr>
                    <td colspan="2"><input type="submit" value="Save"></td>
                </tr>
            </table>
        </form:form>
    </div>
</body>
</html>
```

Click the **Edit** hyperlink next to a customer in the home page, the edit customer form should appear like this:



Edit Customer

ID: 1
Name: Adam K
Email: adamk@gmail.com
Address: Singapore
Save

Code Delete Customer Feature

To implement the delete customer feature, add the following code to the `CustomerController` class:

```
@RequestMapping("/delete")
public String deleteCustomerForm(@RequestParam long id) {
    customerService.delete(id);
    return "redirect:/";
}
```

Click the **Delete** link next to a customer in the home page, it will be deleted and the list is refreshed.

Code Search Customer Feature

Finally, implement the search feature that allows the user to search for customers by typing a keyword. The search function looks for matching keywords in either three fields name, email or address so we need to write a custom method in the `CustomerRepository` interface like this:

```
package com.wipro.customer;

import java.util.List;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

public interface CustomerRepository extends CrudRepository<Customer, Long> {

    @Query(value = "SELECT c FROM Customer c WHERE c.name LIKE '%' || :keyword || '%'"
            + " OR c.email LIKE '%' || :keyword || '%'"
            + " OR c.address LIKE '%' || :keyword || '%'")
    public List<Customer> search(@Param("keyword") String keyword);
}
```

The `search()` method is just an abstract method annotated with the `@Query` annotation. The search query is JPA query.

Then, add the following method in the `CustomerService` class:

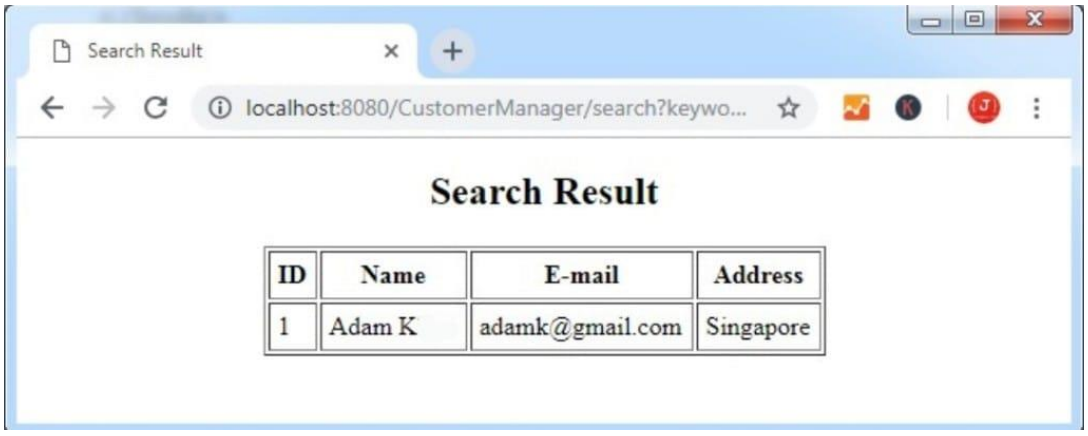
```
public List<Customer> search(String keyword) {
    return repo.search(keyword);
}
```

Implement the handler method in the controller class as follows:

```java
@RequestMapping("/search")
public ModelAndView search(@RequestParam String keyword) {
    List<Customer> result = customerService.search(keyword);
    ModelAndView mav = new ModelAndView("search");
    mav.addObject("result", result);

    return mav;
}
```

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Search Result</title>
</head>
<body>
<div align="center">
    <h2>Search Result</h2>
    <table border="1" cellpadding="5">
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>E-mail</th>
            <th>Address</th>
        </tr>
        <c:forEach items="${result}" var="customer">
        <tr>
            <td>${customer.id}</td>
            <td>${customer.name}</td>
            <td>${customer.email}</td>
            <td>${customer.address}</td>
        </tr>
        </c:forEach>
    </table>
</div>
</body>
</html>
```

To test the search function, type a keyword into the search box in the home page, and hit Enter. You should see the search result page looks like this:



Search Result

ID	Name	E-mail	Address
1	Adam K	adamk@gmail.com	Singapore