Spring Boot + Thymeleaf CRUD Example

Thymeleaf is a modern server-side Java template engine for both web and standalone environments. Thymeleaf able to process HTML, XML, Javascript, CSS, even plain text. It has modules for Spring Framework, and is widely used in Spring based Projects.

In this tutorial, we will learn on how to build a simple Spring Boot application with Thymeleaf as server side templating engine.

Notes: This tutorial goal is to introduce basic CRUD application with **Spring Boot** + **Thymeleaf**. In this tutorial, we will put aside cosmetics things like responsive website design, bootstrap, etc. We just focused on **Thymeleaf** in **Spring Boot** project.

Start a Spring Boot Project

First refer to **Scaffolding Spring Boot Application** to generate your Spring Boot application with (at least) these five dependencies:

- **Spring Web**
  Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- **Spring Data JPA**
  Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

- **Thymeleaf**
  A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.

- **Lombok**
  Java annotation library which helps to reduce boilerplate code.

- **PostgreSQL Driver**
  A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.

The starter for Thymeleaf is `spring-boot-starter-thymeleaf` . In our maven's pom.xml, the dependencies will be like:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

We will use contact database that we create in Spring Boot + JPA/Hibernate + PostgreSQL RESTful CRUD API Example article, please refer to PostgreSQL Configuration section.

ContactApplication is the main class of our application. Please refer the same article for three custom exception classes:

- BadResourceException
- ResourceAlreadyExistsException
- ResourceNotFoundException

Our model, `Contact` class is as following:

Contact.java

```
package com.dariawan.contactapp.domain;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
import lombok.Getter;
import lombok.Setter;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;
import org.springframework.validation.annotation.Validated;
@Validated
@Entity
@Table(name = "contact")
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Getter
@Setter
public class Contact implements Serializable {
private static final long serialVersionUID = 4048798961366546485L;
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

```
private Long id;
@NotBlank
@Size(max = 100)
private String name;
@Pattern(regexp ="^\\+?[0-9. ()-]{7,25}$", message = "Phone number")
@Size(max = 25)
private String phone;
@Email(message = "Email Address")
@Size(max = 100)
private String email;
@Size(max = 50)
private String address1;
@Size(max = 50)
private String address2;
@Size(max = 50)
private String address3;
@Size(max = 20)
private String postalCode;
@Column(length = 4000)
private String note;
}
```

And our repository class, `ContactRepository` :

ContactRepository.java

```
package com.dariawan.contactapp.repository;
import com.dariawan.contactapp.domain.Contact;
import org.springframework.data.jpa.repository.JpaSpecificationExecutor;
import org.springframework.data.repository.PagingAndSortingRepository;
public interface ContactRepository extends PagingAndSortingRepository<Contact, Long>,
JpaSpecificationExecutor<Contact> {
}
```

And in service layer, `ContactService` :

ContactService.java

```
package com.dariawan.contactapp.service;
import com.dariawan.contactapp.domain.Contact;
import com.dariawan.contactapp.exception.BadResourceException;
import com.dariawan.contactapp.exception.ResourceAlreadyExistsException;
import com.dariawan.contactapp.exception.ResourceNotFoundException;
import com.dariawan.contactapp.repository.ContactRepository;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;
import org.springframework.util.StringUtils;
@Service
public class ContactService {
@Autowired
private ContactRepository contactRepository;
private boolean existsById(Long id) {
return contactRepository.existsById(id);
}
public Contact findById(Long id) throws ResourceNotFoundException {
Contact contact = contactRepository.findById(id).orElse(null);
if (contact==null) {
throw new ResourceNotFoundException("Cannot find Contact with id: " + id);
}
else return contact;
}
public List<Contact> findAll(int pageNumber, int rowPerPage) {
List<Contact> contacts = new ArrayList<>();
Pageable sortedByIdAsc = PageRequest.of(pageNumber - 1, rowPerPage,
Sort.by("id").ascending());
contactRepository.findAll(sortedByIdAsc).forEach(contacts::add);
return contacts;
}
public Contact save(Contact contact) throws BadResourceException, ResourceAlreadyExistsException {
if (!StringUtils.isEmpty(contact.getName())) {
if (contact.getId() != null && existsById(contact.getId())) {
throw new ResourceAlreadyExistsException("Contact with id: " + contact.getId() +
" already exists");
}
```

```
return contactRepository.save(contact);
}
else {
BadResourceException exc = new BadResourceException("Failed to save contact");
exc.addErrorMessage("Contact is null or empty");
throw exc;
}
}
public void update(Contact contact)
throws BadResourceException, ResourceNotFoundException {
if (!StringUtils.isEmpty(contact.getName())) {
if (!existsById(contact.getId())) {
throw new ResourceNotFoundException("Cannot find Contact with id: " + contact.getId());
}
contactRepository.save(contact);
}
else {
BadResourceException exc = new BadResourceException("Failed to save contact");
exc.addErrorMessage("Contact is null or empty");
throw exc;
}
}
public void deleteById(Long id) throws ResourceNotFoundException {
if (!existsById(id)) {
throw new ResourceNotFoundException("Cannot find contact with id: " + id);
}
else {
contactRepository.deleteById(id);
}
}
public Long count() {
return contactRepository.count();
}
}
```

Adding Controller

Next, one of the main part of this tutorial - `ContactController` :

ContactController.java

```
package com.dariawan.contactapp.controller;
import com.dariawan.contactapp.domain.Contact;
import com.dariawan.contactapp.exception.ResourceNotFoundException;
import com.dariawan.contactapp.service.ContactService;
import java.util.List;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
@Controller
public class ContactController {
private final Logger logger = LoggerFactory.getLogger(this.getClass());
private final int ROW_PER_PAGE = 5;
@Autowired
private ContactService contactService;
@Value("${msg.title}")
private String title;
@GetMapping(value = {"/", "/index"})
public String index(Model model) { ... }
@GetMapping(value = "/contacts")
public String getContacts(Model model,
@RequestParam(value = "page", defaultValue = "1") int pageNumber) { ... }
@GetMapping(value = "/contacts/{contactId}")
public String getContactById(Model model, @PathVariable long contactId) { ... }
@GetMapping(value = {"/contacts/add"})
public String showAddContact(Model model) { ... }
@PostMapping(value = "/contacts/add")
public String addContact(Model model,
@ModelAttribute("contact") Contact contact) { ... }
@GetMapping(value = {"/contacts/{contactId}/edit"})
public String showEditContact(Model model, @PathVariable long contactId) { ... }
@PostMapping(value = {"/contacts/{contactId}/edit"})
```

```
public String updateContact(Model model,
@PathVariable long contactId,
@ModelAttribute("contact") Contact contact) { ... }
@GetMapping(value = {"/contacts/{contactId}/delete"})
public String showDeleteContactById(
Model model, @PathVariable long contactId) { ... }
@PostMapping(value = {"/contacts/{contactId}/delete"})
public String deleteContactById(
Model model, @PathVariable long contactId) { ... }
}
```

Let's check our controller item by item:

Index Page

The index page or welcome page is a simple page with the title of application and link to contacts page.

```
@Value("${msg.title}")
private String title;
@GetMapping(value = {"/", "/index"})
public String index(Model model) {
model.addAttribute("title", title);
return "index";
}
```

In function index(...) we're adding title attribute to the `Model` so that they can be accessed from the template. The function returning String, which is the template name which will be used to render the response. The template that will be rendered in this function is `index.html` which is available in Thymeleaf default templates location in `src/main/resources/templates/`

index.html

```
<!DOCTYPE HTML>
```

```
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8" />
<title th:utext="${title}" />
<link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1 th:utext="${title}" />
<a th:href="@{/contacts}">Contact List</a>
<br/><br/>
<div>Copyright © dariawan.com</div>
</body>
</html>
```

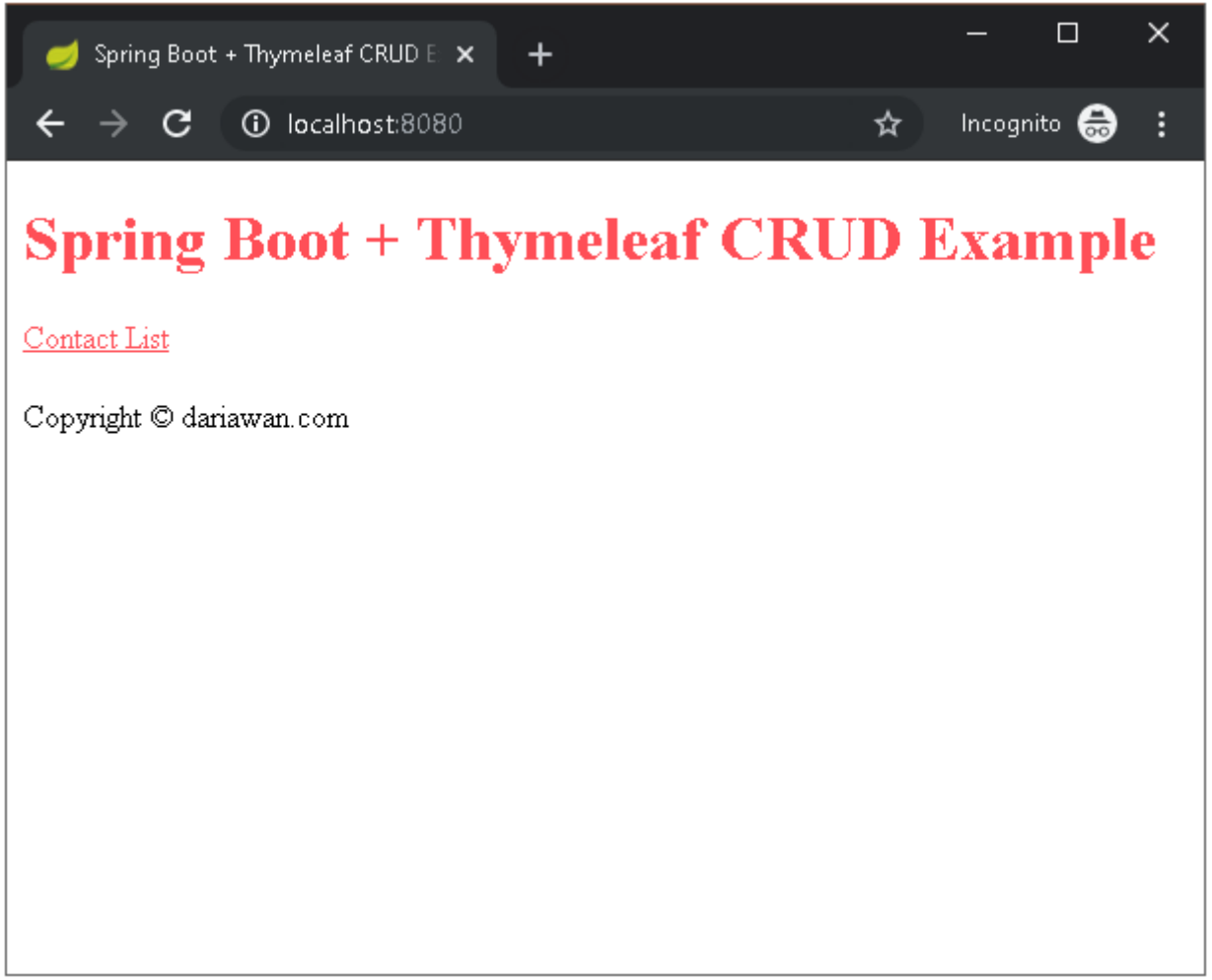
The attribute `th:utext="..."` (and `th:text="..."`) is known as Thymeleaf Standard Dialect, with two different features. As in `<h1 th:utext="${title}" />` example above:

- The `th:utext` attribute, which evaluates its value expression and sets the result of this evaluation as the body of the `h1`.
- The `${title}` expression, specifying that the text to be used by the `th:utext` attribute should be the `title` attribute of the `Model`.

The title attribute is extracted from property file with the `@Value` annotation of `msg.title`. Here the value in `application.properties`:

```
msg.title=Spring Boot + Thymeleaf CRUD Example
```

Here the result in `http://localhost:8080`



http://localhost:8080 (Index Page)

Clicking the "Contact List" link will bring us to contacts page.

Contacts Page

Contacts page will show list of contacts in paged mode (per five records)

```
@GetMapping(value = "/contacts")
public String getContacts(Model model,
```

```
@RequestParam(value = "page", defaultValue = "1") int pageNumber) {
List<Contact> contacts = contactService.findAll(pageNumber, ROW_PER_PAGE);
long count = contactService.count();
boolean hasPrev = pageNumber > 1;
boolean hasNext = (pageNumber * ROW_PER_PAGE) < count;
model.addAttribute("contacts", contacts);
model.addAttribute("hasPrev", hasPrev);
model.addAttribute("prev", pageNumber - 1);
model.addAttribute("hasNext", hasNext);
model.addAttribute("next", pageNumber + 1);
return "contact-list";
}
```

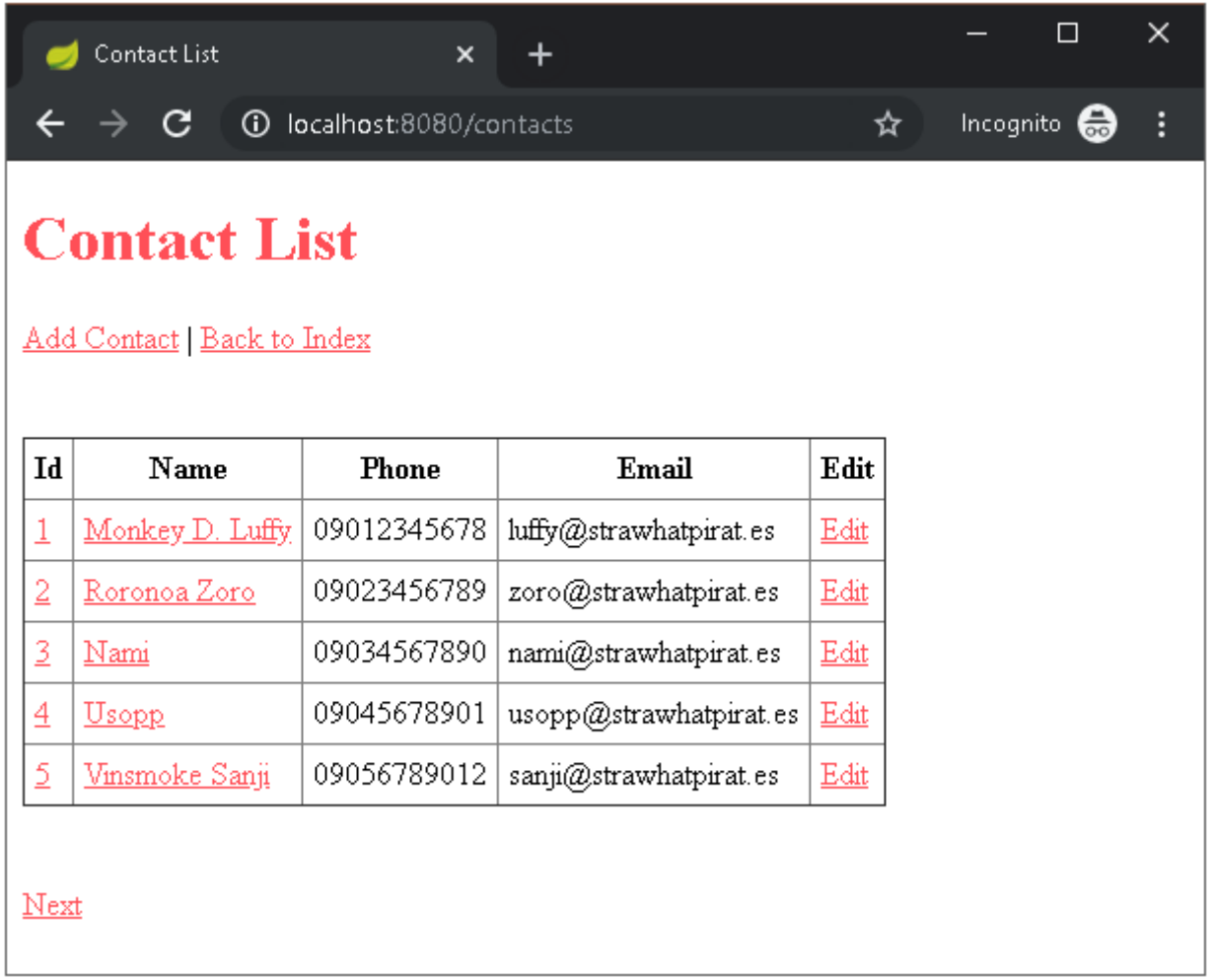
the controller then will render `contact-list.html`

contact-list.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8" />
<title>Contact List</title>
<link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1>Contact List</h1>
<div>
<nobr>
<a th:href="@{/contacts/add}">Add Contact</a> |
<a th:href="@{/}">Back to Index</a>
</nobr>
</div>
<br/><br/>
<div>
<table border="1">
<tr>
<th>Id</th>
<th>Name</th>
<th>Phone</th>
```

```
<th>Email</th>
<th>Edit</th>
</tr>
<tr th:each ="contact : ${contacts}">
<td><a th:href="@{/contacts/{contactId}(contactId=${contact.id})}" th:utext="${contact.id}">...</a></td>
<td><a th:href="@{/contacts/{contactId}(contactId=${contact.id})}" th:utext="${contact.name}">...</a></td>
<td th:utext="${contact.phone}">...</td>
<td th:utext="${contact.email}">...</td>
<td><a th:href="@{/contacts/{contactId}/edit(contactId=${contact.id})}">Edit</a></td>
</tr>
</table>
</div>
<br/><br/>
<div>
<nobr>
<span th:if="${hasPrev}"><a th:href="@{/contacts?page={prev}(prev=${prev})}">Prev</a>   </span>
<span th:if="${hasNext}"><a th:href="@{/contacts?page={next}(next=${next})}">Next</a></span>
</nobr>
</div>
</body>
</html>
```

Here the result of `http://localhost:8080/contacts` :



http://localhost:8080/contacts (Contacts Page)

Clicking the "id" and "name" link will lead us to Contact Page, and clicking the "edit" link will lead to Edit Contact Page.

Add Contact Page is available by clicking "Add Contact" link.

Edit and Add Contact Page

For Add and Edit Contact Page, we will using a similar scenarios:

1. `GET` request to show/render the page, represented by functions `showAddContact(...)` and `showEditContact(...)`

2. `POST` request to save the contact data to the server, represented by functions `addContact(...)` and `updateContact(...)`

```
@GetMapping(value = {"/contacts/add"})
public String showAddContact(Model model) {
Contact contact = new Contact();
model.addAttribute("add", true);
model.addAttribute("contact", contact);
return "contact-edit";
}
@PostMapping(value = "/contacts/add")
public String addContact(Model model,
@ModelAttribute("contact") Contact contact) {
try {
Contact newContact = contactService.save(contact);
return "redirect:/contacts/" + String.valueOf(newContact.getId());
} catch (Exception ex) {
// log exception first,
// then show error
String errorMessage = ex.getMessage();
logger.error(errorMessage);
model.addAttribute("errorMessage", errorMessage);
//model.addAttribute("contact", contact);
model.addAttribute("add", true);
return "contact-edit";
}
}
@GetMapping(value = {"/contacts/{contactId}/edit"})
public String showEditContact(Model model, @PathVariable long contactId) {
Contact contact = null;
try {
contact = contactService.findById(contactId);
} catch (ResourceNotFoundException ex) {
model.addAttribute("errorMessage", "Contact not found");
}
model.addAttribute("add", false);
model.addAttribute("contact", contact);
return "contact-edit";
```

```
}
@PostMapping(value = {"/contacts/{contactId}/edit"})
public String updateContact(Model model,
@PathVariable long contactId,
@ModelAttribute("contact") Contact contact) {
try {
contact.setId(contactId);
contactService.update(contact);
return "redirect:/contacts/" + String.valueOf(contact.getId());
} catch (Exception ex) {
// log exception first,
// then show error
String errorMessage = ex.getMessage();
logger.error(errorMessage);
model.addAttribute("errorMessage", errorMessage);
model.addAttribute("add", false);
return "contact-edit";
}
}
```

For `GET` request, both functions will render `contact-edit.html` :

contact-edit.html

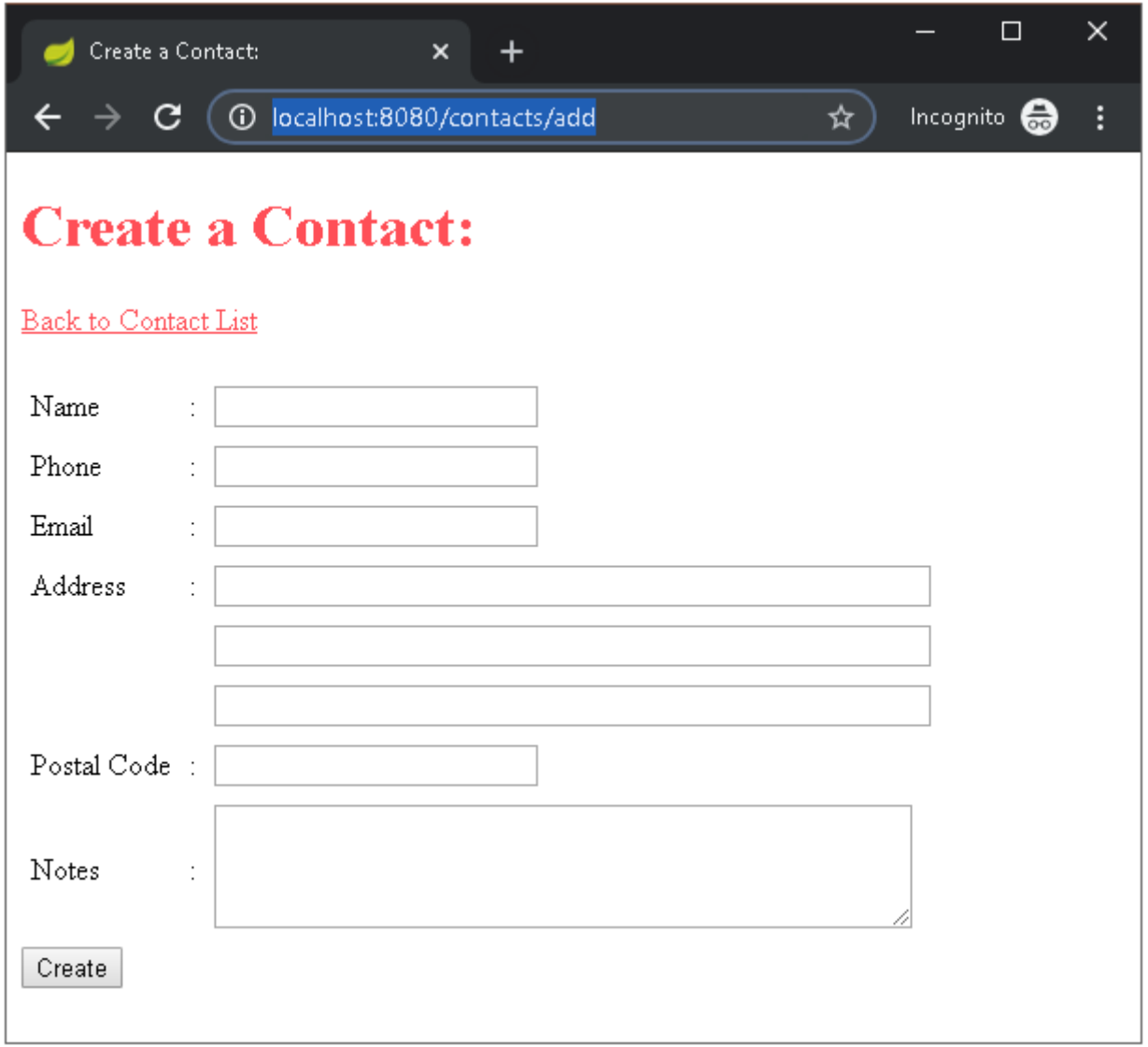
```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8" />
<title th:text="${add} ? 'Create a Contact' : 'Edit a Contact'" />
<link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1 th:text="${add} ? 'Create a Contact:' : 'Edit a Contact:'" />
<a th:href="@{/contacts}">Back to Contact List</a>
<br/><br/>
<form th:action="${add} ? @{/contacts/add} : @{/contacts/{contactId}/edit(contactId=${contact.id})}"
th:object="${contact}" method="POST">
<table border="0">
<tr th:if="${contact.id}">
```

```
<td>ID</td>
<td>:</td>
<td th:utext="${contact.id}">...</td>
</tr>
<tr>
<td>Name</td>
<td>:</td>
<td><input type="text" th:field="*{name}" /></td>
</tr>
<tr>
<td>Phone</td>
<td>:</td>
<td><input type="text" th:field="*{phone}" /></td>
</tr>
<tr>
<td>Email</td>
<td>:</td>
<td><input type="text" th:field="*{email}" /></td>
</tr>
<tr>
<td>Address</td>
<td>:</td>
<td><input type="text" th:field="*{address1}" size="50" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td><input type="text" th:field="*{address2}" size="50" /></td>
</tr>
<tr>
<td></td>
<td></td>
<td><input type="text" th:field="*{address3}" size="50" /></td>
</tr>
<tr>
<td>Postal Code</td>
<td>:</td>
<td><input type="text" th:field="*{postalCode}" /></td>
</tr>
<tr>
<td>Notes</td>
```

```
<td>:</td>
<td><textarea th:field="*{note}" rows="4" cols="50" /></td>
</tr>
</table>
<input type="submit" th:value="${add} ? 'Create' : 'Update'" />
</form>
<br/>
<!-- Check if errorMessage is not null and not empty -->
<div th:if="${errorMessage}" th:utext="${errorMessage}" class="error" />
</body>
</html>
```

From the controller and html above, you can see that attribute `add` is used to control if the page is in "add mode" or "edit mode".

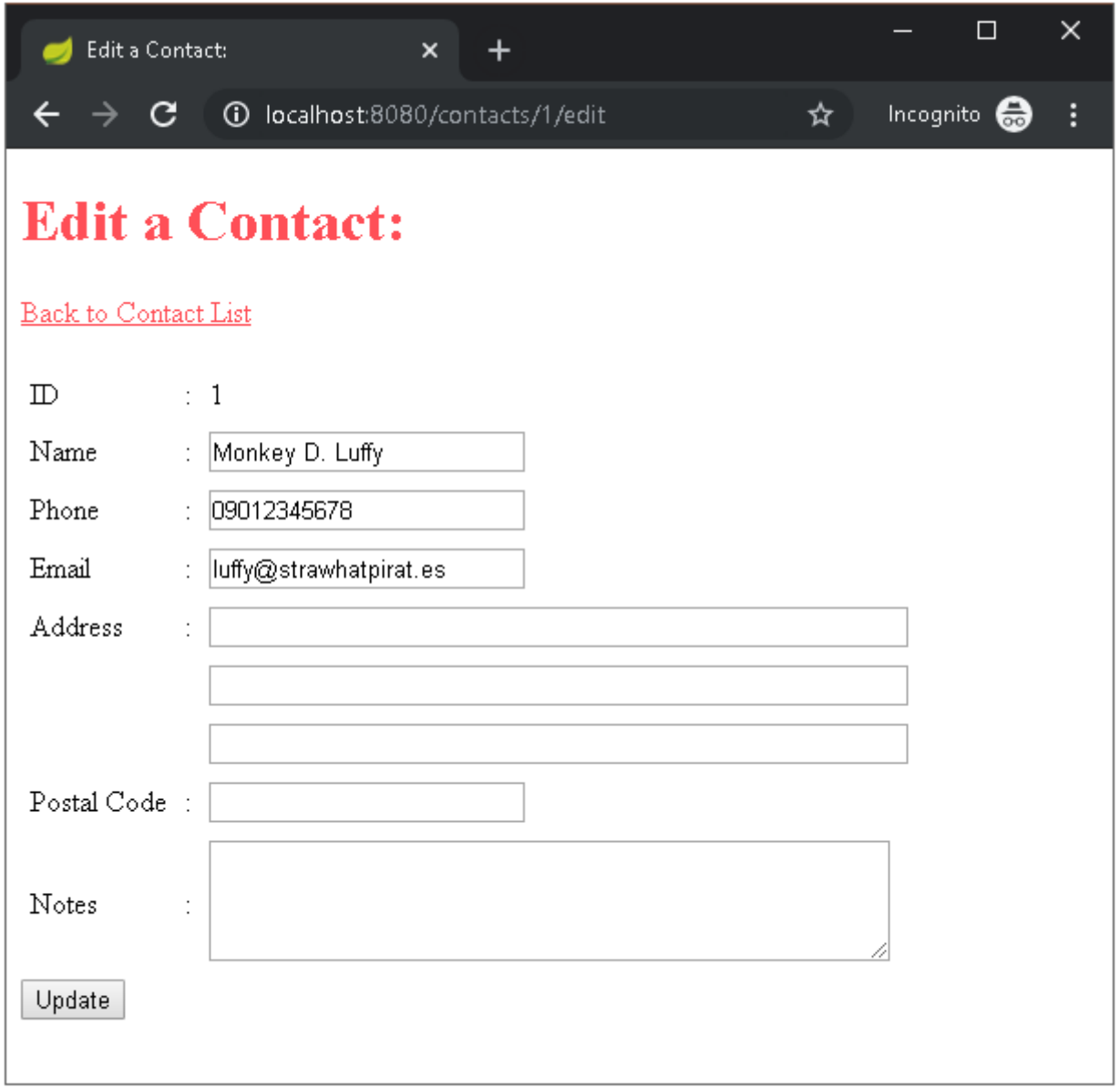
Below is screenshot of Add Contact Page that available in `http://localhost:8080/contacts/add` :



http://localhost:8080/contacts/add (Add Contact Page)

Upon successful add, the controller will redirect to Contact Page to view new created contact.

And below is Edit Contact Page, which as example available in `http://localhost:8080/contacts/1/edit` – for contact with id=1:



http://localhost:8080/contacts/1/edit (Edit Contact Page)

Upon successful update, the controller will redirect to Contact Page to view updated contact.

Contact Page

Contact Page used to show contact in readonly mode. From this page, user can decide to "Edit" or "Delete" contact

```
@GetMapping(value = "/contacts/{contactId}")
public String getContactById(Model model, @PathVariable long contactId) {
Contact contact = null;
try {
contact = contactService.findById(contactId);
} catch (ResourceNotFoundException ex) {
model.addAttribute("errorMessage", "Contact not found");
}
model.addAttribute("contact", contact);
return "contact";
}
```

the controller then will render `contact.html` :

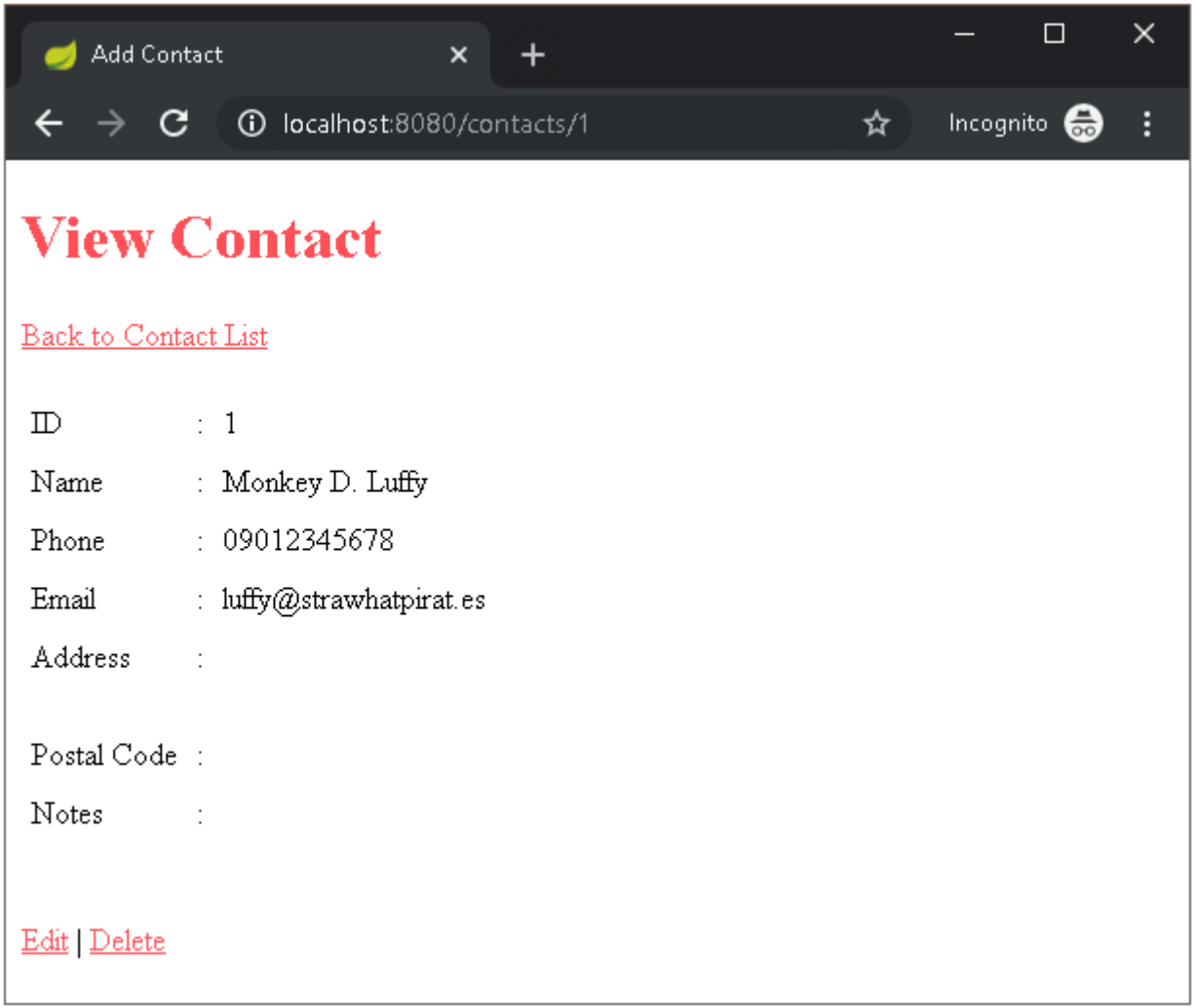
contact.html

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8" />
<title>View Contact</title>
<link rel="stylesheet" type="text/css" th:href="@{/css/style.css}"/>
</head>
<body>
<h1>View Contact</h1>
<a th:href="@{/contacts}">Back to Contact List</a>
<br/><br/>
<div th:if="${contact}">
<table border="0">
<tr>
<td>ID</td>
<td>:</td>
<td th:utext="${contact.id}">...</td>
</tr>
<tr>
<td>Name</td>
```

```
<td>:</td>
<td th:utext="${contact.name}">...</td>
</tr>
<tr>
<td>Phone</td>
<td>:</td>
<td th:utext="${contact.phone}">...</td>
</tr>
<tr>
<td>Email</td>
<td>:</td>
<td th:utext="${contact.email}">...</td>
</tr>
<tr>
<td>Address</td>
<td>:</td>
<td th:utext="${contact.address1}">...</td>
</tr>
<tr>
<td></td>
<td></td>
<td th:utext="${contact.address2}">...</td>
</tr>
<tr>
<td></td>
<td></td>
<td th:utext="${contact.address3}">...</td>
</tr>
<tr>
<td>Postal Code</td>
<td>:</td>
<td th:utext="${contact.postalCode}">...</td>
</tr>
<tr>
<td>Notes</td>
<td>:</td>
<td th:utext="${contact.note}">...</td>
</tr>
</table>
<br/><br/>
<div th:if="not ${allowDelete}">
```

```
<a th:href="@{/contacts/{contactId}/edit(contactId=${contact.id})}">Edit</a> |
<a th:href="@{/contacts/{contactId}/delete(contactId=${contact.id})}">Delete</a>
</div>
<form th:if="${allowDelete}" th:action="@{/contacts/{contactId}/delete(contactId=${contact.id})}" method="POST">
Delete this contact? <input type="submit" th:value="Yes" />
</form>
</div>
<div th:if="${errorMessage}" th:utext="${errorMessage}" class="error" />
</body>
</html>
```

`http://localhost:8080/contacts/1` rendered in browser:



http://localhost:8080/contacts/1 (Contact Page)

If user choose Delete, it will lead to Delete Contact Page.

Delete Contact Page

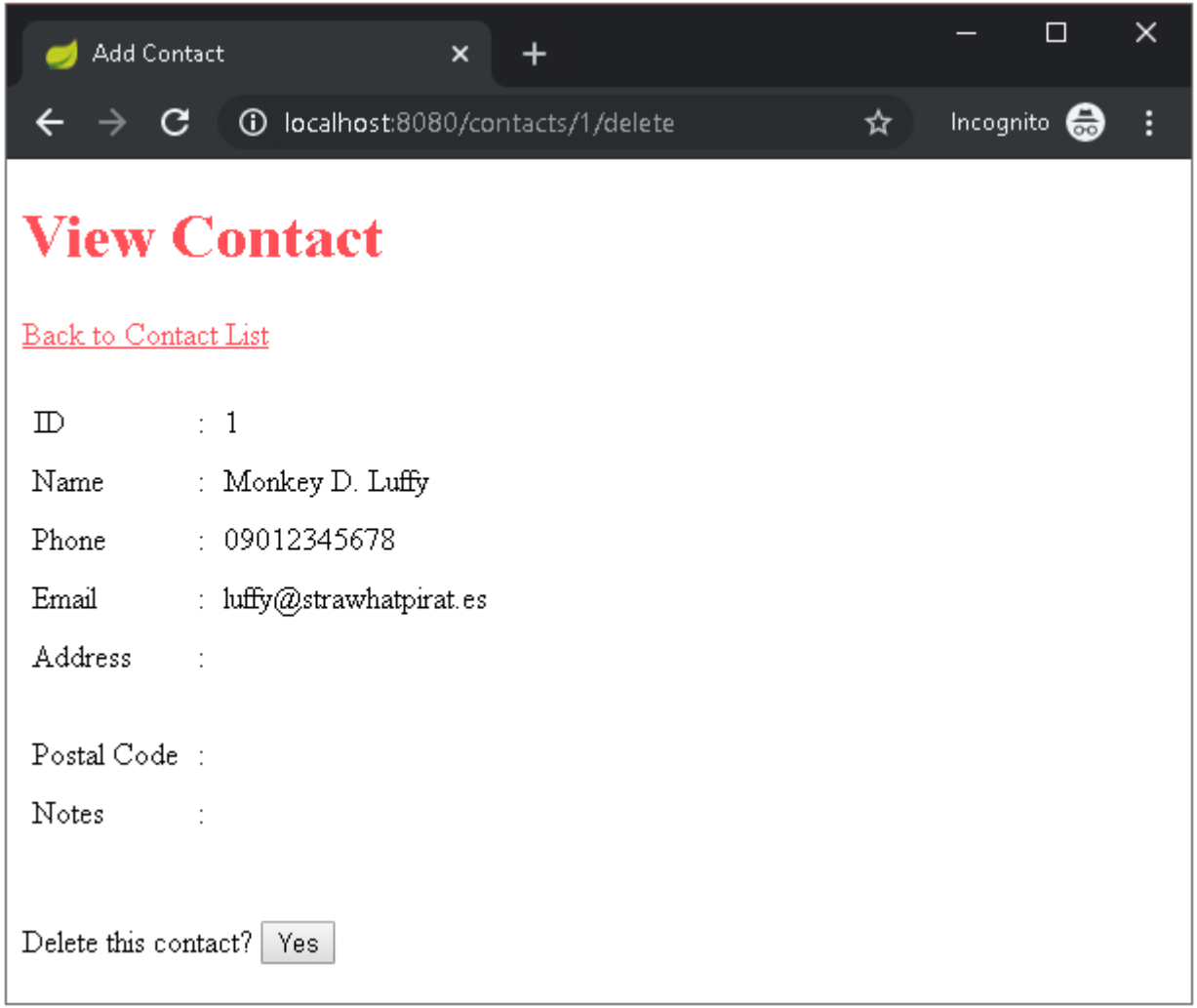
Delete Contact Page using same scenario as Add/Edit Contact Page:

1. `GET` request to show/render the page, represented by functions `showDeleteContactById(...)` to confirm deletion

2. `POST` request to delete contact from the server, represented by functions `deleteContactById(...)`

```
@GetMapping(value = {"/contacts/{contactId}/delete"})
public String showDeleteContactById(
Model model, @PathVariable long contactId) {
Contact contact = null;
try {
contact = contactService.findById(contactId);
} catch (ResourceNotFoundException ex) {
model.addAttribute("errorMessage", "Contact not found");
}
model.addAttribute("allowDelete", true);
model.addAttribute("contact", contact);
return "contact";
}
@PostMapping(value = {"/contacts/{contactId}/delete"})
public String deleteContactById(
Model model, @PathVariable long contactId) {
try {
contactService.deleteById(contactId);
return "redirect:/contacts";
} catch (ResourceNotFoundException ex) {
String errorMessage = ex.getMessage();
logger.error(errorMessage);
model.addAttribute("errorMessage", errorMessage);
return "contact";
}
}
```

Confirm deletion in `http://localhost:8080/contacts/1/delete` :



http://localhost:8080/contacts/1/delete (Delete Contact Page)

Static Files

Static resources default folder is in `\src\main\resources\static\` . The css used for this example is available in `css\style.css`

style.css

```
h1, h2 {
```

```
color:#ff4f57;
}
a {
color: #ff4f57;
}
table {
border-collapse: collapse;
}
table th, table td {
padding: 5px;
}
.error {
color: red;
font-style: italic;
}
```

Final Project Structure

At the end, our project structure will be similar like this:

```
spring-boot-thymeleaf-example
|    .gitignore
|    HELP.md
|    mvnw
|    mvnw.cmd
|    pom.xml
|
|---.mvn
|    |---wrapper
|            maven-wrapper.jar
|            maven-wrapper.properties
|            MavenWrapperDownloader.java
|
|---src
     |---main
     |    |---java
     |    |    |---com
     |    |        |---dariawan
     |    |            |---contactapp
     |    |                |   ContactApplication.java
```

```
    |   |                   |
    |   |                   |---controller
    |   |                   |       ContactController.java
    |   |                   |
    |   |                   |---domain
    |   |                   |       Contact.java
    |   |                   |
    |   |                   |---exception
    |   |                   |       BadResourceException.java
    |   |                   |       ResourceAlreadyExistsException.java
    |   |                   |       ResourceNotFoundException.java
    |   |                   |
    |   |                   |---repository
    |   |                   |       ContactRepository.java
    |   |                   |
    |   |                   |---service
    |   |                   |       ContactService.java
    |   |
    |   |---resources
    |   |       application.properties
    |   |
    |   |---static
    |   |   |---css
    |   |   |       style.css
    |   |
    |   |---templates
    |   |       contact-edit.html
    |   |       contact-list.html
    |   |       contact.html
    |   |       index.html
    |
    |---sql
    |       contact.sql
    |
    |---test
        |---java
            |---com
                |---dariawan
                    |---contactapp
                        |   ContactApplicationTests.java
                        |
                        |---service
                        |       ContactServiceJPATest.java
```

There are some small test files similar like in our previous articles.

Happy Coding!

Liked this Tutorial? Share it on Social media!

Twitter Facebook Linkedin Reddit WhatsApp

This article is part of Getting Started With Spring Boot Series.

Other articles in this series:

Spring Boot Quick Start
Spring Boot Web Application Example
Spring Boot Auto Configuration
Spring Boot Starter
Spring Boot Developer Tools
Spring Boot + JPA/Hibernate + PostgreSQL RESTful CRUD API Example
Spring Boot RESTful Web Services CRUD Example
Documenting Spring Boot REST API with Swagger