COMP S380F Web Applications: Design and Development
Lab 9: Spring Boot, Spring MVC

We will apply Spring MVC to develop a simple customer support application. It allows customers to create a ticket, which contains the customer name, a subject, a body text, and *multiple* file attachments (optional). The following topics will be covered:

- Enabling file upload in Spring MVC
- URI template variable in the URL & @PathVariable in Controller's method argument
- URI template variable with regular expression
- URL redirect using RedirectView or String prefix "redirect:"
- Creating a custom downloading view

Task 1: Build the ticket list page of the customer support application

1. Create a Gradle Spring Boot Project, as follows:
   - Project name: **Lab09**
   - Type: **Gradle - Groovy**
   - Group & Package name: **hkmu.comps380f**
   - Packaging: **Jar**
   - Spring Boot version: **3.2.3**
   - Dependencies:
     - Web > **Spring Web**
     - Ops > **Spring Boot Actuator**
2. In **`build.gradle`**, add the following dependencies, and reload the project in the Gradle window.

```
dependencies {
    compileOnly 'org.apache.tomcat.embed:tomcat-embed-jasper'
    implementation 'jakarta.servlet.jsp.jstl:jakarta.servlet.jsp.jstl-api'
    implementation 'org.glassfish.web:jakarta.servlet.jsp.jstl'
    implementation 'jakarta.el:jakarta.el-api'
    implementation 'org.apache.commons:commons-lang3'
    …
}
```

   The dependency `tomcat-embed-jasper` is for JSP development, and `commons-lang3` is for generating random alphanumeric strings for identifiers of the uploaded files. Note that we also remove version number from the dependencies and use Spring Boot's dependency management.

3. Enable JSP development by creating the **`webapp`** folder (with a ***blue dot***, see Lecture 7 Slide 11).

4. Add the following property values to the property file **`/resources/application.properties`**:

```
server.servlet.context-path=/Lab09

server.servlet.session.timeout=30m
server.servlet.session.cookie.http-only=true
server.servlet.session.tracking-modes=cookie

spring.mvc.view.prefix=/WEB-INF/jsp/
spring.mvc.view.suffix=.jsp

spring.servlet.multipart.enabled=true
spring.servlet.multipart.max-file-size=20MB
spring.servlet.multipart.max-request-size=40MB
spring.servlet.multipart.file-size-threshold=5MB
spring.servlet.multipart.location=${java.io.tmpdir}
```

The property `spring.servlet.multipart.enabled=true` enables **`DispatcherServlet`** to support file uploads, and creates a Spring bean **`StandardServletMultipartResolver`** (in addition to the **`InternalResourceViewResolver`** for JSP view resolver) to handle file uploads. We can also set the following properties:

- `spring.servlet.multipart.max-file-size` specifies the maximum file size of an uploaded file. Larger files are not accepted by the web app.
- `spring.servlet.multipart.max-request-size` specifies the maximum size of a request, regardless of the number of files to upload. Larger requests are not served.
- `spring.servlet.multipart.file-size-threshold` specifies the minimum file size for an uploaded file to be put in a temporary file directory. Smaller files are only kept in memory.
- `spring.servlet.multipart.location` tells the web container which directory is used to store temporary files. We use `${java.io.tmpdir}` to define the temporary file directory such that it works for different operating systems.

5. Create the file **`/webapp/WEB-INF/jsp/base.jspf`** as the prelude of every JSP page:

```
<%@ taglib prefix="c" uri="jakarta.tags.core" %>
<%@ taglib prefix="fn" uri="jakarta.tags.functions" %>
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

6. Add the following Spring bean for common JSP config to the main class **`Lab09Application`**:

```
@Bean
public ConfigurableServletWebServerFactory configurableServletWebServerFactory() {
    return new TomcatServletWebServerFactory() {
        @Override
        protected void postProcessContext(Context context) {
            super.postProcessContext(context);
            JspPropertyGroup jspPropertyGroup = new JspPropertyGroup();
            jspPropertyGroup.addUrlPattern("*.jsp");
            jspPropertyGroup.addUrlPattern("*.jspf");
            jspPropertyGroup.setPageEncoding("UTF-8");
            jspPropertyGroup.setScriptingInvalid("true");
            jspPropertyGroup.addIncludePrelude("/WEB-INF/jsp/base.jspf");
            jspPropertyGroup.setTrimWhitespace("true");
            jspPropertyGroup.setDefaultContentType("text/html");
            JspPropertyGroupDescriptorImpl jspPropertyGroupDescriptor
                    = new JspPropertyGroupDescriptorImpl(jspPropertyGroup);
            context.setJspConfigDescriptor(
                    new JspConfigDescriptorImpl(
                            Collections.singletonList(jspPropertyGroupDescriptor),
                            Collections.emptyList()));
        }
    };
}
```

We are now ready to define the models, views, and controllers for Spring MVC.

Model (package: `hkmu.comps380f.model`)

- **Attachment.java**: This POJO stores a file attachment. Complete the getters and setters.

```
public class Attachment {
    private String id;
    private String name;
    private String mimeContentType;
    private byte[] contents;

    // Getters and Setters of id, name, mimeContentType, contents
}
```

- **Ticket.java**: This POJO stores a ticket. It has a property a property `attachments` mapping the String Attachment id to an Attachment object. Complete the getters and setters.

```
public class Ticket {
    private long id;
    private String customerName;
    private String subject;
    private String body;
    private Map<String, Attachment> attachments = new ConcurrentHashMap<>();

    // Getters and Setters of id, customerName, subject, body (not attachments)

    public Attachment getAttachment(String name) {
        return this.attachments.get(name);
    }

    public Collection<Attachment> getAttachments() {
        return this.attachments.values();
    }

    public void addAttachment(Attachment attachment) {
        this.attachments.put(attachment.getId(), attachment);
    }

    public int getNumberOfAttachments() {
        return this.attachments.size();
    }
}
```

Controller (package: `hkmu.comps380f.controller`)

IndexController.java: This controller redirects all HTML requests for the base URL to the application's list page.

```
@Controller
public class IndexController {
    @GetMapping("/")
    public String index() {
        return "redirect:/ticket/list";
    }
}
```

TicketController.java: This Controller provides business logic for the customer support application.

```
@Controller
@RequestMapping("/ticket")
public class TicketController {
    private volatile long TICKET_ID_SEQUENCE = 1;
    private Map<Long, Ticket> ticketDatabase = new ConcurrentHashMap<>();

    // Controller methods, Form-backing object, ...
}
```

- Using `@RequestMapping("/ticket")` on the controller class, all its controller methods will be mapped to URLs beginning with <base URL>/ticket/.
- There can be multiple threads serving the requests for the same servlet. For thread-safety, the `volatile` keyword is a lightweight `sychronized` that can be used on variables (but not classes/methods) and can guarantee the threads always read the most updated value.

Now, we define the controller method for listing all tickets. A GET request to the URL <base URL>/ticket or <base URL>/ticket/list will map to the JSP view for listing all the tickets:

```
@GetMapping(value = {"", "/list"})
public String list(ModelMap model) {
    model.addAttribute("ticketDatabase", ticketDatabase);
    return "list";
}
```

View (directory: `/WEB-INF/jsp`)

- **list.jsp**: This JSP page lists all the tickets.

```
<!DOCTYPE html>
<html>
<head>
    <title>Customer Support</title>
</head>
<body>
<h2>Tickets</h2>
<a href="<c:url value="/ticket/create" />">Create a Ticket</a><br/><br/>
<c:choose>
    <c:when test="${fn:length(ticketDatabase) == 0}">
        <i>There are no tickets in the system.</i>
    </c:when>
    <c:otherwise>
        <c:forEach items="${ticketDatabase}" var="entry">
            Ticket ${entry.key}:
            <a href="<c:url value="/ticket/view/${entry.key}" />">
                <c:out value="${entry.value.subject}"/></a>
            (customer: <c:out value="${entry.value.customerName}"/>)<br />
        </c:forEach>
    </c:otherwise>
</c:choose>
</body>
</html>
```

  - `<c:out value="..."/>` escapes reserved XML characters. Using it on user-supplied content helps protect the application from cross-site scripting and various injection attacks, and helps prevent unexpected special characters breaking the functionality of the application. It is a good habit of always using `<c:out>` to display user-supplied texts.

7. Run the web application to check that the list page can be displayed.

Task 2: Build the remaining parts of the customer support application

View (directory: `/WEB-INF/jsp`)

- **add.jsp**: This JSP page is for creating a ticket. To support uploading multiple files, we add the attribute `multiple="multiple"` in the tag `<input type="file" … />`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Customer Support</title>
</head>
<body>
<h2>Create a Ticket</h2>
<form:form method="POST" enctype="multipart/form-data" modelAttribute="ticketForm">
    <form:label path="customerName">Customer Name</form:label><br/>
    <form:input type="text" path="customerName"/><br/><br/>
    <form:label path="subject">Subject</form:label><br/>
    <form:input type="text" path="subject"/><br/><br/>
    <form:label path="body">Body</form:label><br/>
    <form:textarea path="body" rows="5" cols="30"/><br/><br/>
    <b>Attachments</b><br/>
    <input type="file" name="attachments" multiple="multiple"/><br/><br/>
    <input type="submit" value="Submit"/>
</form:form>
</body>
</html>
```

  - Note that `<form:form>` has the attribute `enctype="multipart/form-data"`, which is required when you are using HTML forms that have a file upload control.

- We have used the Spring MVC's tag library `form` in this HTML form. This allows Spring to tie HTML form parameters to the form-backing object `ticketForm` of class **`Form`**, which will be defined in the controller **`TicketController`**.

- **view.jsp**: This JSP page shows the details of a particular ticket.

```
<!DOCTYPE html>
<html>
<head>
    <title>Customer Support</title>
</head>
<body>
<h2>Ticket #${ticketId}: <c:out value="${ticket.subject}"/></h2>
<i>Customer Name - <c:out value="${ticket.customerName}"/></i><br/><br/>
<c:out value="${ticket.body}"/><br/><br/>
<c:if test="${ticket.numberOfAttachments > 0}">
    Attachments:
    <c:forEach items="${ticket.attachments}" var="attachment" varStatus="status">
        <c:if test="${!status.first}">, </c:if>
        <a href="<c:url value="/ticket/${ticketId}/attachment/${attachment.id}" />">
            <c:out value="${attachment.name}"/></a>
    </c:forEach><br/><br/>
</c:if>
<a href="<c:url value="/ticket" />">Return to list tickets</a>
</body>
</html>
```

  - The `varStatus` attribute in `<c:forEach>` defines a variable indicating the current status of the iteration. `status.first` is true if the current iteration is the first iteration; otherwise, it is false. Other available status includes `begin`, `end`, `step`, `index`, `count`, `last`.

Controller (package: `hkmu.comps380f.controller`)

TicketController.java: Now, we define more controller methods in the controller **`TicketController.`**

1. **Creating a ticket with an HTML form**: We define the `Form` class for the form-backing object of the above HTML form. Then, we define the model attribute `ticketForm` for a new `Form` object, which will be used in the JSP view **`add.jsp`**. Complete the getters and setters.

```
@GetMapping("/create")
public ModelAndView create() {
    return new ModelAndView("add", "ticketForm", new Form());
}

public static class Form {
    private String customerName;
    private String subject;
    private String body;
    private List<MultipartFile> attachments;

    // Getters and Setters of customerName, subject, body, attachments
}
```

2. **Creating a ticket after receiving submitted form data:** The function `getNextTicketId()` is used to generate a unique ticket ID for a new ticket (keyword `synchronized` is for thread-safety).

```
@PostMapping("/create")
public View create(Form form) throws IOException {
    Ticket ticket = new Ticket();
    ticket.setId(this.getNextTicketId());
    ticket.setCustomerName(form.getCustomerName());
    ticket.setSubject(form.getSubject());
    ticket.setBody(form.getBody());

    for (MultipartFile filePart : form.getAttachments()) {
        Attachment attachment = new Attachment();
        attachment.setId(RandomStringUtils.randomAlphanumeric(8));
        attachment.setName(filePart.getOriginalFilename());
        attachment.setMimeContentType(filePart.getContentType());
        attachment.setContents(filePart.getBytes());
        if (attachment.getName() != null && attachment.getName().length() > 0
                && attachment.getContents() != null && attachment.getContents().length > 0)
            ticket.addAttachment(attachment);
    }
    this.ticketDatabase.put(ticket.getId(), ticket);
    return new RedirectView("/ticket/view/" + ticket.getId(), true);
}

private synchronized long getNextTicketId() {
    return this.TICKET_ID_SEQUENCE++;
}
```

We add the method argument "`Form form`" in this controller method. Spring will automatically map it to the form-backing object of the class `Form`, which ties to all the HTML form parameters. Recall that arguments in a controller method can be added in any order; Spring will automatically map each argument to the appropriate object.

In **`add.jsp`**, the input control `<input type="file" name="attachments" multiple="multiple"/>` does not use the Spring tag library `form`. Yet the form-backing object can still get the attachments, because the form-backing object has a property of the same name "`attachments`".

When using `RedirectView`, the second boolean argument "`true`" refers to whether the URL pattern is context-relative or not (i.e., relative to the base URL (= *server URL + context root*) or relative to the server URL), so we will redirect the user to visit the URL:

<base URL>/ticket/view/<ticket ID>

Also, the ticket ID is appended to the URL, so we will need to use ***URI template variable*** in the URL pattern of `@GetMapping` in the controller method `view` for viewing a ticket (see below).

3. **Viewing a ticket of a particular ticket ID:** In the URL value of following `@GetMapping`, `ticketId` is an ***URI template variable***. We can use the annotation `@PathVariable("ticketId")` in the method argument to get the value of this URI template variable.

```
@GetMapping("/view/{ticketId}")
public String view(@PathVariable("ticketId") long ticketId,
                   ModelMap model) {
    Ticket ticket = this.ticketDatabase.get(ticketId);
    if (ticket == null) {
        return "redirect:/ticket/list";
    }
    model.addAttribute("ticketId", ticketId);
    model.addAttribute("ticket", ticket);
    return "view";
}
```

An alternative way to redirect the user to an URL is to use the prefix `redirect:` on a URL pattern. If we find that the required ticket does not exist, we will redirect the user to the list page.

4. **Downloading a file attachment in a particular ticket ID:** An URI template variable can be used together with a **regular expression**. If the corresponding part in the URL does not match the regular expression, the URL will not be mapped to the controller method. In the following method, the URI template variable `attachment` must have at least one character (as indicated by the **regular expression** "`.+`").

```
@GetMapping("/{ticketId}/attachment/{attachment:.+}")
public View download(@PathVariable("ticketId") long ticketId,
                     @PathVariable("attachment") String AttachmentId) {
    Ticket ticket = this.ticketDatabase.get(ticketId);
    if (ticket != null) {
        Attachment attachment = ticket.getAttachment(AttachmentId);
        if (attachment != null)
            return new DownloadingView(attachment.getName(),
                    attachment.getMimeContentType(), attachment.getContents());
    }
    return new RedirectView("/ticket/list", true);
}
```

When the attachment can be retrieved, we use a custom view called `DownloadingView`, (see below), which can hand over the file download to the client's browser. This class implements Spring's `View` interface, and we need to override the method `render()` for handing over the file download to the browser. Note the use of `Map<String, ?>`, where `?` means any class.

View class (package: `hkmu.comps380f.view`)

```
public class DownloadingView implements View {
    private final String filename;
    private final String contentType;
    private final byte[] contents;

    public DownloadingView(String filename, String contentType, byte[] contents) {
        this.filename = filename;
        this.contentType = contentType;
        this.contents = contents;
    }

    @Override
    public String getContentType() {
        return this.contentType;
    }

    @Override
    public void render(Map<String, ?> model, HttpServletRequest request,
                       HttpServletResponse response) throws Exception {
        response.setHeader("Content-Disposition", "attachment; filename=" + this.filename);
        response.setContentType("application/octet-stream");

        ServletOutputStream stream = response.getOutputStream();
        stream.write(this.contents);
    }
}
```