COMP S380F Web Applications: Design and Development
Lab 10: Spring Data JPA with Hibernate

We will have exercises on using Spring Data JPA and Hibernate to persist data in relational database. The following topics will be covered:

- Defining entity classes for database tables with one-to-many relationship
- Using automatic JPA repository generated by Spring Data JPA
- Storing file attachments in database

Task 1: Initializing database tables for tickets and attachments using Spring Boot

Download the branch "**lab10**" of the Github repository "**https://github.com/cskeith/380_2024.git**".

1. In **`build.gradle`**, remove "`commons-lang3`" and add the dependencies below. Reload the project.

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
runtimeOnly 'com.h2database:h2'
```

2. In **`application.properties`**, add the following properties.

```
# H2 data source
spring.datasource.url=jdbc:h2:./Data/myDB;AUTO_SERVER=TRUE
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

# Auto-initialize DB tables according to definition of entity classes
spring.jpa.hibernate.ddl-auto=update

# Show Hibernate-generated SQL in console & disable open-in-view, custom error page
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.open-in-view=false
server.error.whitelabel.enabled=false
server.error.include-exception=true
```

3. In package **`hkmu.comps380f.model`**, make **`Attachment`** and **`Ticket`** as entity classes. Note that an entity class requires a **default constructor**, and all annotations are from **`jakarta.persistence`**.

```
@Entity
public class Ticket {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private long id;

    @Column(name = "name")
    private String customerName;
    private String subject;
    private String body;

    @OneToMany(mappedBy = "ticket", fetch = FetchType.EAGER,
            cascade = CascadeType.ALL, orphanRemoval = true)
    @Fetch(FetchMode.SUBSELECT)
    private List<Attachment> attachments = new ArrayList<>();

    // getters and setters of all properties

    public void deleteAttachment(Attachment attachment) {
        attachment.setTicket(null);
        this.attachments.remove(attachment);
    }
}
```

Box 1

```
@Entity
public class Attachment {
    @Id
    @GeneratedValue
    @ColumnDefault("random_uuid()")
    private UUID id;

    @Column(name = "filename")
    private String name;

    @Column(name = "content_type")
    private String mimeContentType;

    @Column(name = "content")
    @Basic(fetch = FetchType.LAZY)
    @Lob
    private byte[] contents;

    @Column(name = "ticket_id", insertable=false, updatable=false)     Box 2
    private long ticketId;

    @ManyToOne
    @JoinColumn(name = "ticket_id")                                     Box 3
    private Ticket ticket;

    // getters and setters of all properties
}
```

One ticket may have many attachments, so the table `ticket` has a one-to-many relationship with the table `attachment`. In JPA terminology, the `Ticket` entity is the owner of the one-to-many relationship.

In the entity class `Ticket`, we use `@OneToMany` to define a list of `Attachment` objects (**Box 1**):

- `mappedBy` indicates that the `Attachment` entity is a ***dependent entity*** owned by the `Ticket` entity in the one-to-many relationship. Its value is the variable name of the `Ticket` object defined in the `Attachment` entity (see **Box 3**).
- `fetch=FetchType.EAGER` indicates that the list of `Attachment` objects will be loaded together when we load the `Ticket` object from the database.
  If this is absent, the default setting is `fetch=FetchType.LAZY`, where the list of `Attachment` objects is loaded on-demand (i.e., lazy loading) when we call the `getAttachments()` method.
- `cascade=CascadeType.ALL` indicates that the persistence will propagate (cascade) all `EntityManager` operations (`PERSIST`, `REMOVE`, `REFRESH`, `MERGE`, `DETACH`) to the relating entities, so when we save a `Ticket` entity to the table `ticket`, all its `Attachment` entities referencing this user are saved to the table `attachment` automatically.
- `orphanRemoval=true` indicates that a disconnected `Attachment` entity is automatically removed from the table `attachment`. This is useful for cleaning up dependent objects (e.g., `Attachment`) that should not exist without a reference from an owner object (e.g., `Ticket`).
- In the annotation `@Fetch`, `FetchMode` defines how Hibernate will fetch the data (by `SELECT`, `JOIN` or `SUBSELECT`). On the other hand, `FetchType` in the annotation `@OneToMany` defines whether Hibernate will load the data eagerly or lazily.

In the entity class `Attachment`, we use `@ManyToOne` to define a `Ticket` object (**Box 3**):

- The table column `ticket_id` in the table `attachment` is a foreign key pointing to the table column `id` in the table `ticket`. We use `@ManyToOne` to define a referenced `Ticket` object.

The @JoinColumn indicates that the Ticket entity is the ***owner entity*** of the one-to-many relationship.

- As the table column ticket_id is used to define both entity class properties ticketId and ticket, we only update the property ticket while the value of ticketId should be set automatically accordingly. Therefore, we use the annotation

  ```
  @Column(name = "ticket_id", insertable=false, updatable=false)
  ```

  on the entity class property ticketId (**Box 2**).
- We use UUIDs as the identifiers of Attachment entity objects. UUIDs are globally unique, so we don't need a centralized component to generate unique ids. In the annotation @ColumnDefault, we use H2's built-in function random_uuid() to generate a random UUID as the default value of the id column.

4. With the entity classes, we add the repositories for Ticket and Attachment to the **dao** package hkmu.comps380f.dao:

```
public interface TicketRepository extends JpaRepository<Ticket, Long> {
}
```

```
public interface AttachmentRepository extends JpaRepository<Attachment, UUID> {
}
```

Spring Data JPA will automatically implement a set of CRUD methods, e.g., count, exists, save, delete, deleteById, deleteAll, findById, findAll, etc.

5. We define two new exception classes in an **exception** package hkmu.comps380f.exception:

```
public class TicketNotFound extends Exception {
    public TicketNotFound(long id) {
        super("Ticket " + id + " does not exist.");
    }
}
```

```
public class AttachmentNotFound extends Exception {
    public AttachmentNotFound(UUID id) {
        super("Attachment " + id + " does not exist.");
    }
}
```

6. We apply the **Controller-Service-Repository** design pattern:
   - **Repository layer:** Repositories provide the persistence logic, i.e., it is responsible for all logic related to saving data to a data store and retrieving saved data from the data store. Typically, **each repository is responsible for a single entity class**. In Spring, we use @Repository to define a repository class. Yet when using Spring Data JPA, we only need to define the interface and the repository objects will be generated accordingly.
   - **Service layer:** Services encapsulate the business logic of the application and consume other services and repositories but do not consume resources in higher application layers like controllers. In Spring, services are annotated with @Service. From a transactional point of view, the execution of a service method from a higher layer (such as a controller) can be thought of as a transactional unit of work. **A service may perform several operations on multiple repositories.**
   - **Controller layer:** Controllers provide the user interface logic, which controls the user interface, uses services for assistance, and prepare the model for presentation in the view. In Spring, controllers are marked with the @Controller annotation.

We create the following service class in the **dao** package `hkmu.comps380f.dao`.

```
@Service
public class TicketService {
    @Resource
    private TicketRepository tRepo;

    @Resource
    private AttachmentRepository aRepo;

    @Transactional
    public List<Ticket> getTickets() {
        return tRepo.findAll();
    }

    @Transactional
    public Ticket getTicket(long id)
            throws TicketNotFound {
        Ticket ticket = tRepo.findById(id).orElse(null);
        if (ticket == null) {
            throw new TicketNotFound(id);
        }
        return ticket;
    }

    @Transactional
    public Attachment getAttachment(long ticketId, UUID attachmentId)
            throws TicketNotFound, AttachmentNotFound {
        Ticket ticket = tRepo.findById(ticketId).orElse(null);
        if (ticket == null) {
            throw new TicketNotFound(ticketId);
        }
        Attachment attachment = aRepo.findById(attachmentId).orElse(null);
        if (attachment == null) {
            throw new AttachmentNotFound(attachmentId);
        }
        return attachment;
    }

    @Transactional(rollbackFor = TicketNotFound.class)
    public void delete(long id) throws TicketNotFound {
        Ticket deletedTicket = tRepo.findById(id).orElse(null);
        if (deletedTicket == null) {
            throw new TicketNotFound(id);
        }
        tRepo.delete(deletedTicket);
    }

    @Transactional(rollbackFor = AttachmentNotFound.class)
    public void deleteAttachment(long ticketId, UUID attachmentId)
            throws TicketNotFound, AttachmentNotFound {
        Ticket ticket = tRepo.findById(ticketId).orElse(null);
        if (ticket == null) {
            throw new TicketNotFound(ticketId);
        }
        for (Attachment attachment : ticket.getAttachments()) {
            if (attachment.getId().equals(attachmentId)) {
                ticket.deleteAttachment(attachment);
                tRepo.save(ticket);
                return;
            }
        }
        throw new AttachmentNotFound(attachmentId);
    }
    @Transactional
    public long createTicket(String customerName, String subject,
                             String body, List<MultipartFile> attachments)
            throws IOException {
        Ticket ticket = new Ticket();
        ticket.setCustomerName(customerName);
        ticket.setSubject(subject);
        ticket.setBody(body);

        for (MultipartFile filePart : attachments) {
            Attachment attachment = new Attachment();
            attachment.setName(filePart.getOriginalFilename());
            attachment.setMimeContentType(filePart.getContentType());
            attachment.setContents(filePart.getBytes());
            attachment.setTicket(ticket);
            if (attachment.getName() != null && attachment.getName().length() > 0
                    && attachment.getContents() != null
                    && attachment.getContents().length > 0) {
                ticket.getAttachments().add(attachment);
            }
        }
        Ticket savedTicket = tRepo.save(ticket);
        return savedTicket.getId();
    }
}
```

The annotation `@Transactional(rollbackFor=...)` indicates that the transaction would be rolled back if the specified exception is thrown. Note that `@Transactional` is from the package `org.springframework.transaction.annotation`.

Due to `cascade=CascadeType.ALL` for the `attachments` list variable in the `Ticket` entity, we do not need to save the `Attachment` entity object using the `save()` function explicitly. When we save a `Ticket` entity object, all its `Attachment` objects will be saved to the table `attachment` with the `ticket_id` column equal to the generated `id` of the saved `Ticket` object.

7. The last step is to update the controller **`TicketController`** and the JSP view pages.

```
@Controller
@RequestMapping("/ticket")
public class TicketController {

    @Resource
    private TicketService tService;

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

a) **Listing all the tickets:** We use the `tService` to get all `Ticket` objects in the table `ticket`.

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

Your task: We update the JSP view **list.jsp** accordingly.

b) **Creating a ticket**: We use `tService` to save a `Ticket` object in the table `ticket`.

```
@PostMapping("/create")
public View create(Form form) throws IOException {
    long ticketId = tService.createTicket(form.getCustomerName(),
            form.getSubject(), form.getBody(), form.getAttachments());
    return new RedirectView("/ticket/view/" + ticketId, true);
}
```

c) **Viewing a particular ticket:** We use `tService` to get the `Ticket` with a particular ticket ID.

```
@GetMapping("/view/{ticketId}")
public String view(@PathVariable("ticketId") long ticketId,
                   ModelMap model)
        throws TicketNotFound {
    Ticket ticket = tService.getTicket(ticketId);
    model.addAttribute("ticketId", ticketId);
    model.addAttribute("ticket", ticket);
    return "view";
}
```

Your task: We update the JSP view **view.jsp** accordingly.

d) **Downloading a file attachment:** We use `tService` to get a particular attachment.

```
@GetMapping("/{ticketId}/attachment/{attachment:.+}")
public View download(@PathVariable("ticketId") long ticketId,
                     @PathVariable("attachment") UUID attachmentId)
        throws TicketNotFound, AttachmentNotFound {
    Attachment attachment = tService.getAttachment(ticketId, attachmentId);
    return new DownloadingView(attachment.getName(),
               attachment.getMimeContentType(), attachment.getContents());
}
```

e) **Custom error page:** Note that in **`application.properties`**, we have disabled the default error page by setting the property `server.error.whitelabel.enabled=false`. The property `server.error.include-exception=true` allows us to have information of the `Exception`. We can create a custom error page with a view name "**`error`**", i.e., **`error.jsp`**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Customer Support</title>
</head>
<body>
<h2>Error page</h2>
<c:choose>
    <c:when test="${empty message}">
        <p>Something went wrong.</p>
        <ul>
            <li>Status Code: ${status}</li>
            <li>Exception: ${exception}</li>
        </ul>
    </c:when>
    <c:otherwise>
        <p>${message}</p>
    </c:otherwise>
</c:choose>
<a href="<c:url value="/ticket" />">Return to list tickets</a>
</body>
</html>
```

Note that in **TicketController**, we can add an error handler to handle TicketNotFound.class and AttachmentNotFound.class and show the exception message in the above error view page:

```
@ExceptionHandler({TicketNotFound.class, AttachmentNotFound.class})
public ModelAndView error(Exception e) {
    return new ModelAndView("error", "message", e.getMessage());
}
```

8. Run the web app. Follow Lecture 8 Slides 28-29 to view the database table contents.

Task 2: Enabling deleting tickets and attachments

1. In **TicketController**, add the following two methods:

a) **Deleting a particular ticket:** We use tService to delete a Ticket object with a particular ticket ID. After a Ticket object is deleted, all its attachments are also automatically deleted.

```
@GetMapping("/delete/{ticketId}")
public String deleteTicket(@PathVariable("ticketId") long ticketId)
        throws TicketNotFound {
    tService.delete(ticketId);
    return "redirect:/ticket/list";
}
```

b) **Deleting a particular attachment:** We use tService to delete an attachment in a ticket.

```
@GetMapping("/{ticketId}/delete/{attachment:.+}")
public String deleteAttachment(@PathVariable("ticketId") long ticketId,
                               @PathVariable("attachment") UUID attachmentId)
        throws TicketNotFound, AttachmentNotFound {
    tService.deleteAttachment(ticketId, attachmentId);
    return "redirect:/ticket/view/" + ticketId;
}
```

2. Update the view **list.jsp**, as shown in red:

```
(customer: <c:out value="${entry.customerName}"/>)
[<a href="<c:url value="/ticket/delete/${entry.id}" />">Delete</a>]<br />
```

3. Update the view **view.jsp**, as shown in red:

```
<h2>Ticket #${ticketId}: <c:out value="${ticket.subject}"/></h2>
[<a href="<c:url value="/ticket/delete/${ticket.id}" />">Delete</a>]<br/><br/>
<i>Customer Name - <c:out value="${ticket.customerName}"/></i><br/><br/>
```

```
<a href="<c:url value="/ticket/${ticketId}/attachment/${attachment.id}" />">
    <c:out value="${attachment.name}"/></a>
[<a href="<c:url value="/ticket/${ticketId}/delete/${attachment.id}" />">Delete</a>]
```

Task 3: Initializing the database tables using SQL files

1. In IntelliJ's database view, drop the two tables ticket and attachment.

2. Restart the web app, and record the Hibernate-generated SQL create statements.

3. Add a new ticket (without any attachments). In IntelliJ's database view, right-click on the table `ticket` and select **Import/Export** > **Export Data to File**. Choose the **Extractor** "SQL Inserts" and then click **Copy to Clipboard**. Record the SQL insert statement.
4. In **`application.properties`**, update the following properties.

```
# Initialize DB tables using SQL files
spring.jpa.hibernate.ddl-auto=none
spring.sql.init.mode=always
spring.sql.init.schema-locations=classpath:sql/schema.sql
spring.sql.init.data-locations=classpath:sql/data.sql
```

5. Based on your recorded information, you should be able to come up with SQL statements for **`schema.sql`** and **`data.sql`** similar to the following:

/resources/sql/schema.sql:

```
create table if not exists ticket (
    id bigint generated by default as identity,
    body varchar(255),
    name varchar(255),
    subject varchar(255),
    primary key (id)
);
create table if not exists attachment (
    id uuid default random_uuid() not null,
    content blob,
    content_type varchar(255),
    filename varchar(255),
    ticket_id bigint,
    primary key (id),
    foreign key (ticket_id) references ticket
);
```

/resources/sql/data.sql:

```
insert into ticket (body, name, subject)
    values ('This is a test message.', 'Keith', 'Test subject');
```

6. Drop the two tables `ticket` and `attachment`. Restart the web app.