This documentation is intended for readers with a background of web-based application development using Spring MVC and Hibernate. The purpose of this document is to describe common patterns used to build components in the applications. These components are: the data model, hibernate mapping, service layer, and controller layer.

This project stores data in these tables: ACCOUNT, INGREDIENT\_TYPE, INGREDIENTS, LIKED\_RECIPE, RECIPE\_INGREDIENT, and RECIPE table.

The ACCOUNT table is intended to record registered user of the website. The INGREDIENT\_TYPE table is intended to record ingredient of recipes recorded in the database. The INGREDIENT table is intended to record the amount of ingredients for recipes recorded in the database. The LIKED\_RECIPE table is intended to record the number of user who saved a recipe into their liked-recipe list. The RECIPE table is intended to record recipes. The RECIPE\_INGREDIENT table is intended as a link table between the RECIPE table and INGREDIENT table. The DDL script for each table is stored under src/main/resources.

**MODELLING THE DATA MODEL WITH JPA ANNOTATIONS**

Relationships between these tables are modelled using JPA. To do this, we annotate the POJO with an @Entity annotation, which means that this POJO is an entity class. The @Table annotation defines the table name in the database that this POJO is mapped to. All getter methods in a POJO is annotated with an @column annotation, which means that this getter method is mapped to a column in the table. Every POJO class has a version attribute, and the associated getter method is annotated with @Version annotation. The intention is to instruct Hibernate, that we would like to use an optimistic locking mechanism, using the version attribute as a control. control. Every time Hibernate updates a record, it will compare the version of the entity instance to that of the record in the database. If both versions are the same, it means that no one updated the data before, and Hibernate will update the data and increment the version column. However, if the version is not the same, it means that someone has updated the record before, and Hibernate will throw a StaleObjectStateException exception, which Spring will translate to HibernateOptimisticLockingFailureException.

There are three types of relationship between the tables. They are one-to-many, many-to-many, and one-to-one. The following sections describe the usage of JPA annotations to model all these relationships.

**MODELLING A ONE-TO-MANY RELATIONSHIP**

Every recipe has at least one recipe or many ingredients. This relationship is modelled with three tables: the RECIPE table, the RECIPE\_INGREDIENT table, and the INGREDIENT table. The association between each recipe and its ingredients are captured in the recipe ingredient table. We decided not to store a foreign key of an ingredient into the RECIPE table to keep each row in the RECIPE table to store one recipe. This way displaying all recipes in the first page of the website can be attained using a simple select query: select \* from recipe.

To model the relationship between the RECIPE table, the RECIPE\_INGREDIENT table, and the INGREDIENT table we use two kind of relationship. A one-to-many relationship between the RECIPE table and the RECIPE\_INGREDIENT table is modelled using @OneToMany annotation and @JoinTable on the getIngredients method. We also specified a @Cascade annotation with a persist option and a merge option. This mean that whenever we save a Recipe, all related ingredients will be saved into the INGREDIENT table.

**MODELLING A MANY-TO-MANY RELATIONSHIP**

Registered users are allowed to like recipes. Every user can have zero or many recipes that he likes. Therefore, every recipe can be like by zero or many users. This relationship is modelled in the website with a many-to-many mapping. A many-to-many relationship requires a join table, which is the LIKEDRECIPE table.

The Account class holds information on a user including recipe that the user likes, which is modelled using a java.util.Set. The getter method that returns this set is annotated with the @ManyToMany annotation.

@ManyToMany(fetch = FetchType.LAZY)

@JoinTable(name = "LikedRecipe", joinColumns = @JoinColumn(name = "email"), inverseJoinColumns = @JoinColumn(name = "recipeId"))

public Set<Recipe> getLikedRecipes() {

return likedRecipes;

}

The method is also annotated with @JoinTable annotation to indicate that the LIKEDRECIPE table records recipes that a user likes. The LIKEDRECIPE table has a foreign key constraint that references email column in the ACCOUNT table. This relationship is modelled using the joinColumns in the @ManyToMany annotation. The LIKEDRECIPE table also has a foreign key constraint that references the recipeId column in the recipe table. This relationship is modelled using the inverseJoinColumns in the @ManyToMany annotation.

**MODELLING A MANY-TO-ONE RELATIONSHIP**

Ingredients of recipes in the website are stored in two tables. The INGREDIENT table stores the amount and unit of measurement of the ingredients. The names of the ingredients are stored in the INGRDIENTTYPE table under the NAME column. The table enforced a UNIQUE constraint for the name column. This ensures that INGREDIENTTYPE table stores unique ingredient names.

Administrator of the website can select ingredients stored in the INGREDIENTTYPE table when he create new recipes using the add recipe page. Storing the ingredient names in the INGREDIENTTYPE table makes it easy to display unique ingredient names on that page.

One or many ingredients recorded in the INGREDIENT table can share a single ingredient name recorded in the INGREDIENTTYPE table. This relationship is modelled using @ManyToOne annotation.

@ManyToOne (cascade = { CascadeType.ALL })

@JoinColumn(name="ingredientType")

@NotNull(message="{validation.ingredient.NotEmpty.message}")

public IngredientType getIngredientType() {

return ingredientType;

}

@JoinColumn annotation specifies a column which a foreign key constraints references the INGREDIENTTYPE table. @NotNull annotation specifies that the ingredientType cannot be empty.

**ORM MAPPING USING JPA**

These tables are useless unless we can use them to store recipes and display them on the website. To do that, we need to use a framework capable of storing and retrieving data from these tables. This framework is called JPA. JPA must know how to connect to the database. This is achieved by recording connection details to our database in a configuration file. This includes the URL, the name, and password of the database. This configuration file is called datasource-tx-jpa.xml in src\main\resources\ folder.

Once JPA knows how to connect to a database, JPA accesses the database using an EntityManager. In this project, we are using an entity manager provided by spring framework named LocalContainerEntityManagerFactoryBean. This manager manages entities created for accessing recipe data from the database.

There are three ways for managing transaction. Two of them are for declarative transaction management, with one using Java annotations and the other using XML configuration. The third option is managing transactions programmatically. This website uses Java annotation to manage transactions because annotation makes the application easier to trace and maintain.

To enable annotation support for transaction management, a <tx:annotation-driven> tag is added in an XML configuration file: C:\source\Recipe\src\main\resources\datasource-tx-jpa.xml. And since, the website uses JPA to access the database, a JPA transaction manager is defined as a bean in the xml file:

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

There are two data sources recorded in the XML configuration file. Each data source is configured for 2 different profiles: dev and mysql. The dev profile is used for unit test purposes. This profile defines a datasource that uses H2 database. This database is an in memory database. The intention of using an in memory database is to avoid having to clean up test data from production database. Mysql profile defines a connection to a MySQL database. The website uses this database to store recipes.

**SERVICE LAYER**

This layer consists of a number of java classes intended to provide methods that allow the controller layer to access database.

Every service class in this project is annotated with @Service annotation. This intention is to identify that this class is a Spring component that provide business services to another layer. The @Repository annotation indicates that the class contains data access logic and instructs Spring to translate the vendor-specific exceptions to Spring’s DataAccessException hierarchy. The entity manager is injected by using @PersistenceContext annotation.

Every service class in this project is annotated with the @Transactional annotation. One of that class is the RecipeServiceImpl. This class has a number of find mehods. All these methods are annotated with @Transactional(readOnly=true), because these methods only read data and do not modify them. The intention is to optimize these methods.

**Querying data from the database**

The service layer uses JPQL to query the database. One example of a JPQL is named queries defined in entity classes such as the Account class. The account class is intended to map columns in the ACCOUNT table. This class has a named query:

@NamedQuery(name = "Account.findByEmail", query = "select a from Account a where a.email = :email")

This named query is used to retrieve user accounts with a particular email address by the AccountServiceImpl.

@Transactional(readOnly=true)

public Account findByEmail(String name) {

TypedQuery<Account> typedQuery = entityManager.createNamedQuery("Account.findByEmail",Account.class) typedQuery.setParameter("email", name);

return typedQuery.getSingleResult();

}

Beside named queries, the AccountServiceImpl class also passes JPQL into the entity manager using untyped result queries. For example:

public void unlikeARecipe(String email, Long recipeId) {

Query query = entityManager.createQuery("delete from LikedRecipe where recipeId = :recipeId and email = :email");

query.setParameter("recipeId", recipeId);

query.setParameter("email", email);

query.executeUpdate();

entityManager.clear();

}

Untyped result queries and named queries are not the only two flavour of JPQL that this project uses. Constructor expression is another form of JPQL that this project uses. The RecipeServiceImpl uses a constructor expression that pulls out recipe information for the home page of the website.

select new com.safe.stack.domain.RecipeSummary(r.id, r.name, r.author, r.diet, (select count(\*) from LikedRecipe l where r.id = l.recipeId) as numOfLikes, r.authorLink, r.picture, (select count(\*) from LikedRecipe l where r.id = l.recipeId and l.email =:arg0) as likedByUser) from Recipe r;

The above query instructs JPA to construct a RecipeSummary object.

**Inserting data into the database**

The project uses Spring Data to save data into a database. The central interface that this project uses is Repository. An example of that Respository is com.safe.stack.repository. LikedRecipeRepository. This interface extends org.springframework.data.repository.CrudRepository. The intention of this class is to expose methods convenient for saving data. For example:

@Override

public void likeARecipe(String userName, Long recipeId) {

LikedRecipe likedRecipe = new LikedRecipe();

likedRecipe.setEmail(userName);

likedRecipe.setRecipeId(recipeId);

likedRecipeRepository.save(likedRecipe);

}

This method is intended to persist a LikedRecipe object into the database. A LikedRecipe object is an entity the records data for the LIKED\_RECIPE table.

**CONTROLLER LAYER**

The controller layer handles request from web pages and interact with the service layer. There is one controller in the project, which is the HomeController. This section describes common pattern that the HomeController uses to handle request from web pages. To tell spring that HomeController is a Controller, this class is annotated with @Controller. The @RequestMapping annotation used at the class level indicates URL that HomeController handles.

@RequestMapping annotations used at method level of HomeController indicates URL that a method handles. For Example, @RequestMapping(value = "/searchRecipeByIngredient", method = RequestMethod.POST). This annotation indicates that method annotated with this annotation handles a URL that ends with /searchRecipeByIngredient with POST method.

Some methods in this class are annotated with @PreAuthorize("isAuthenticated'), which prevents a user from accessing URLs handled by that method from being accessed by user who are not logged in.

HomeController also validates user input using JSR-303 validation. This is accomplished using @Valid annotation at parameter in a method. For example, public String signUp(@Valid Account acc ... . The @Valid annotation enables validation when a request binds data into the Account object. When there is an error in the binding result, the errors will be saved into a BindingResult object. These errors canbe displayed using Spring's form tag. For example <form:errors path="email"/>. This tag will display errors resulted from binding an invalid email address to the Account object.

**ENABLING SPRING SECURITY**

In the controller section, it was briefly mentioned that the Controller uses annotation to prevent user from accesing URL that require user to log in. This is achieved by enabling spring security. To do this, a filter is registered in the web.xml file:

<filter>

<filter-name>springSecurityFilterChain</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

This filter is configured to intercept request coming from any URL from the website. This is achieved by recording the url-pattern:

<filter-name>springSecurityFilterChain</filter-name>

<url-pattern>/\*</url-pattern>

<dispatcher>REQUEST</dispatcher>

<dispatcher>FORWARD</dispatcher>

</filter-mapping>

Any requests from users will be authenticated by spring security, and users will not be given access to URL that requires login when they have not logged in to the system. The website uses login framework that comes with Spring Security, and this framework is configured in a separate XML file called security-context.xml.

...

<form-login login-page="/login" authentication-failure-url="/loginfail"

...

The form-login tag defines that the login page URL is /login. The UserService class encapsulates logic for authenticating a user.