Object Oriented Design

Implementation

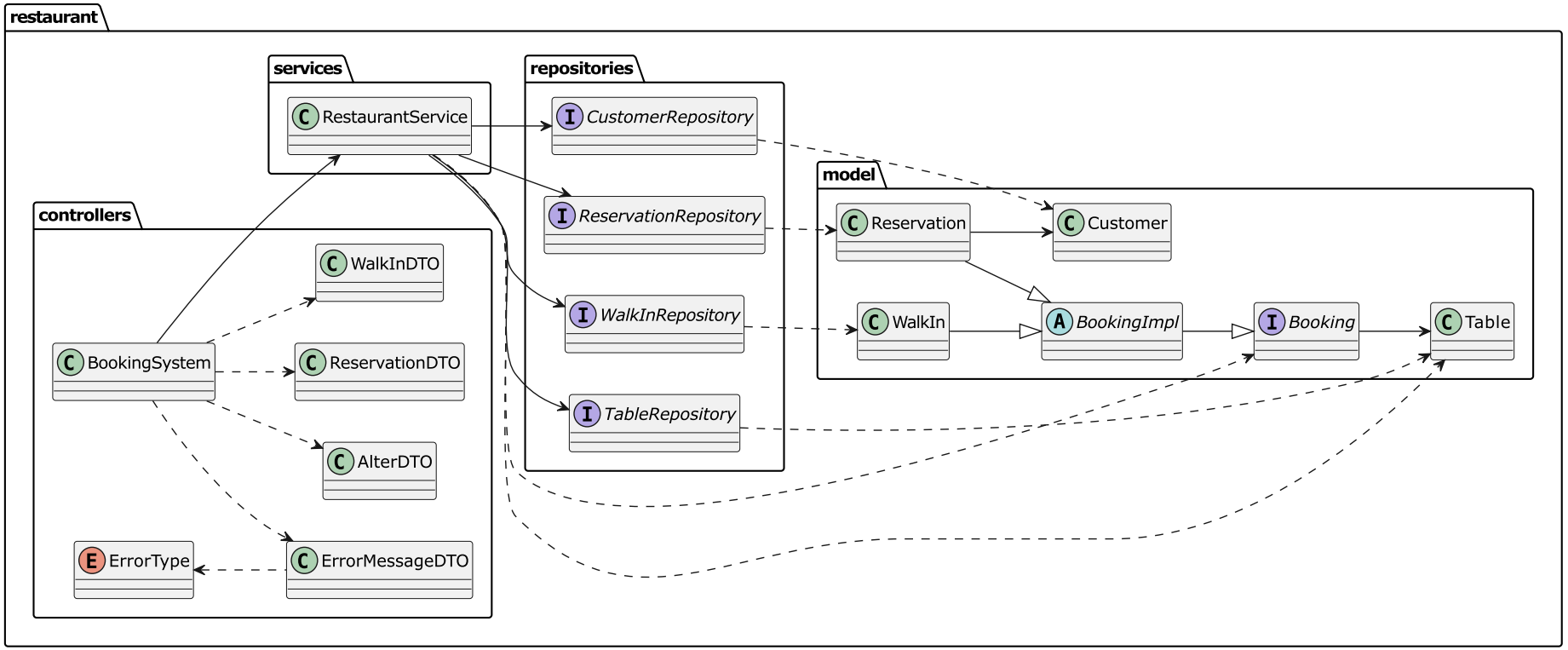
Dr. Seán Russell

School of Computer Science,
University College Dublin

Lecture 09

Table of Contents

Section Contents


restaurant
services
RestaurantService
repositories
CustomerRepository
ReservationRepository
WalkInRepository
TableRepository
model
Reservation
Customer
WalkIn
BookingImpl
Booking
Table
controllers
WalkInDTO
BookingSystem
ReservationDTO
AlterDTO
ErrorType
ErrorMessageDTO

- The dependencies shown in the component diagram impose constraints on the order that components can be created and tested
- Two basic approaches have been described to this
  - **Top-down implementation**
  - **Bottom-up implementation**

- Top-down implementation starts with the higher components and proceeds downwards in the direction of the dependency arrows
- An advantage of this strategy is that the overall design of the system can be tested early in the process
- A disadvantage is that **stubs**, or temporary implementations, need to be created for lower level classes
- These will need to be replaced later by the real implementation as the development progresses

- Bottom-up implementation starts with the lower level classes and proceeds up the diagram
- This approach makes the development and testing of individual components easier
- When a class is implemented, all the classes on which it depends have already been implemented
- This makes it easier to compile and test the class without stubs
- However, this approach runs the risk of postponing a complete executable program until quite a late stage in implementation

- A compromise is to adopt a more iterative approach and think of implementing use cases instead of classes
- With this approach, developers implement those features of each class required to support a single use case
- This use case is then fully tested
- Further use cases are then implemented one by one

Section Contents

- Spring Boot projects can look big and seem to have a lot of detail
- However, we can use a shortcut to create our projects easily
- The site `https://start.spring.io` can be used to create a basic project
- We choose our versions, names and dependencies, then download the created project

Project
○ Gradle Project ● Maven Project

Language
● Java ○ Kotlin ○ Groovy

Spring Boot
○ 3.0.0 (SNAPSHOT) ○ 3.0.0 (RC1) ○ 2.7.6 (SNAPSHOT) ● 2.7.5
○ 2.6.14 (SNAPSHOT) ○ 2.6.13

Project Metadata

Group	com.example
Artifact	demo
Name	demo
Description	Demo project for Spring Boot
Package name	com.example.demo
Packaging	● Jar ○ War
Java	○ 19 ● 17 ○ 11 ○ 8

Dependencies ADD DEPENDENCIES... CTRL + B

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Thymeleaf TEMPLATE ENGINES
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.

Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

H2 Database SQL
Provides a fast in-memory database that supports JDBC API and R2DBC access, with a small (2mb) footprint. Supports embedded and server modes as well as a browser based console application.

Section Contents

- Maven is a tool that can now be used for building and managing any Java-based project
- Maven builds a project using its project object model (POM) and a set of plugins
- We must define the details in XML format in a file named `pom.xml`
- It will download all necessary dependencies and build our project

- The POM contains many details about the project
  - Such as group, name, version, description, Java version
- The part we are mostly concerned with is the dependencies section
- This is likely the part that we will change as we need to

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
↪ "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
  ↪ "http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.0</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>ie.ucd.comp3013j</groupId>
  <artifactId>restaurant</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>restaurant</name>
  <description>Restaurant Example for COMP3013J</description>
  <properties>
    <java.version>17</java.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</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-data-jpa</artifactId>
    </dependency>
```

- There are a lot of commands that can be used to do different tasks using Maven
- We will discuss the most useful ones
  - `mvn clean` - deletes any existing compiled files
  - `mvn compile` - compiles the source code of your project
  - `mvn package` - generates a jar/war file from your project
  - `mvn spring-boot:run` - runs the application

Section Contents

- The generate project has a set layout, we should use this

```
projectfolder
  +- pom.xml
  +- (some maven files)
  +- src
     +- test
     +- main
        +- java
           +- group/and/project/name
              +- JAVA FILES HERE
        +- resources
           +- application.properties
           +- static
           +- templates
              +- mainpage.html
```

- The properties of spring boot plugins can be specified by including them in the properties file
- This can be a properties file (`.properties`) or yaml file (`.yaml`)
- The file should be placed in the resources folder and be named application
- Properties will be specific to an individual plugin

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.username=sa
spring.datasource.password=password
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
```

```
spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driverClassName: org.h2.Driver
    username: sa
    password: password
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
  h2:
    console:
      enabled: 'true'
```

- Much of the features of Spring Boot are based on the use of annotations in our code
- These can be added to classes, constructors, methods, and variables
- Here are some examples:
  - `@SpringBootApplication` - the main class of the application
  - `@Autowired` - Used to automatically insert objects into the correct places
  - `@Controller` - A controller that manages input data and building models for templates
  - `@Entity` - Defines that this class should be persisted in the database

- The `@SpringBootApplication` annotation tells spring boot this is the main class of the application
- This will be created automatically for us
- Spring Boot will scan all Java files in this package and any sub packages looking for annotations and classes that can be autowired
- This should always be in the top most package

```
1  @SpringBootApplication
2  public class RestaurantApplication {
3
4      public static void main(String[] args) {
5          SpringApplication.run(RestaurantApplication.class, args);
6      }
7
8  }
```

- The `@Autowired` annotation allows us to have the correct object inserted automatically for us
- This means that we do not need to think about how the object gets there

```
1  public class RestaurantService {
2      private final CustomerRepository customers;
3      private final ReservationRepository reservations;
4      private final TableRepository tables;
5      private final WalkInRepository walkIns;
6
7      @Autowired
8      public RestaurantService(CustomerRepository c, ReservationRepository r, TableRepository t,
       ↪  WalkInRepository w) {
9          customers = c;
10         reservations = r;
11         tables = t;
12         walkIns = w;
13     }
```

- The `@Controller` annotation simply tells Spring Boot that this class is a controller
- Spring boot will then evaluate the annotations on methods to find mappings
- Examples of mappings would be:
  - `@GetMapping("/")` - A GET request to the site will result in this method being called
  - `@PostMapping("/")` - A POST to the site will result in this method being called
  - `@PostMapping("/delete/")` - A POST to this part of the site will result in this method being called

Section Contents

- Managing persistent data in Spring Boot applications is done through annotations
- The most important is the `@Entity` annotation
- This can be applied to a class and marks it as something that can be persisted in the database
- We can specify the name of the table that will be used in the database
  - E.g. `@Entity(name = "dinner_tables")`

- Designating a class as an entity places some requirements
- The entity class must have a constructor that takes no parameters, though it may have other constructors as well
- You should have getter and setter methods for any instance variables that will be remembered

- Objects in memory can be distinguished by their address, but these will change every execution
- Tables in a database require a primary key to distinguish one row from another
- We can add an identifier to each class to represent the primary key
- Annotations `@Id` and `@GeneratedValue` will tell JPA to manage the values
  - E.g. `@Id @GeneratedValue long id;`

- All instance variables in the class are automatically included in the database
- The `@Column` annotation can be used to change some detail
  - E.g. `@Column(columnDefinition="LONGTEXT")`
  - E.g. `@Column(name="step_order") private int order;`
- This can be useful to prevent problems, to have consistently chosen names, or to apply some logic
  - E.g. `@Column(nullable=false)`

- All properties are also represented in the database
  - A property is a matched getter and setter method and the value associated with them
- You can use the `@Column` annotation on either method
- If you do not want the property to be persisted, or if the method is not actually connected to a property you can use the `@Transient` annotation
- This tells JPA that the value should be ignored

- If the instance variable in our entity class is also another entity, we need to define the relationship
- If the relationship is only in one direction, we can annotate the variable with `@OneToOne`
- This will will add a column containing a foreign key to the database table
- If the relationship is bidirectional (each holds a reference to the other) it is more complicated

- For bidirectional references, we need to add more annotations than just `@OneToOne`
- We should choose which one of the objects will be the owner in the relationship and will contain the foreign key column
- In the owner class, we add the `cascade` value to the annotation, so that saving one object saves the other
- In the other, we add the `mappedBy` value to the annotation and specify the name of the instance variable in the owner class

```
public class A{
  @OneToOne(cascade = CascadeType.
  ↪  ALL)
  private B refToB;
}
```

```
public class B{
  @OneToOne(mappedBy = "refToB")
  private A refToA;
}
```

- A different solution is required if an instance variable is a data structure
- We can use the `@ElementCollection` annotation
  - This is typically used with basic types like Integer and String
  - It can also be used with objects annotated as with the `@Embeddable`

- For more complex data that is represent as an entity, we need to consider the relationship
- Based on this, we can decide to use a `@OneToMany` or `@ManyToMany` annotation
  - This is why we needed to consider these relationships during design

- This shows where one Book can have many Authors

```java
@Entity
public class Book{
  @OneToMany(cascade = CascadeType.ALL)
  public List<Author> authors;
}
```

```java
@Entity
public class Author {
  String name;
}
```

- This can only work if the relationship is one to many

- This shows where one Book can have many Authors, and each Author can have many Books

```java
@Entity
public class Book{
  @ManyToMany(cascade=CascadeType.ALL)
  private List<Author> authors;
  String title;
}
```

```java
@Entity
public class Author {
  @ManyToMany(mappedBy="authors")
  private List<Book> books;
  String name;
}
```

Section Contents

- A Repository is an interface that extends `JpaRepository`
- This defines how a particular class of object will be queried in the database
- A number of methods, and the queries they represent will be available automatically
- Any other queries must be defined by correctly naming our methods

- If we define a repository that extends `JpaRepository<T, Long>`, then we have the following methods
  - `T save(T t);`
  - `Optional<T> findById(Long id);`
  - `List<T> findAll();`
  - `long count();`
  - `void delete(T t);`

- Any other queries that we wish to use must be defined in the interface

```
1  List<Book> findByTitle(String title);
2  List<Book> findByTitleContaining(String title);
3  Optional<Book> findByIsbn(String isbn);
4  List<Customer> findByFeesDueGreaterThan(int feesDue);
5  List<Customer> findByDateOfBirthBetween(LocalDate start, LocalDate end
   ↪  );
6  List<Customer> findByFeesDueGreaterThanAndFeesDueLessThanEqual(int low
   ↪  , int high);
7  long deleteByTitle(String title);
```

Section Contents

- Thymeleaf is a Java based HTML template engine
- This means that we can add special code to HTML files and it will fill in details for us
- We must provide a template (containing HTML and potentially JS code)
- We must add the required data to the Model
- Thymeleaf will create the final HTML and JS that is seen in the browser

- To generate a web page, we must first have a HTML template
  - This will contain special annotations to indicate where values should be filled in
  - These may be simple insertion or something more complex like a for each loop
- Thymeleaf modifies existing HTML tags though the use of additions to attributes
- Any attribute processed by Thymeleaf will start with `th:`
  - E.g. `th:value="${date}"`
  - E.g. `th:href="@{/(date=${yesterday})}"`

- Typically, anything inside a set of curly brackets that is after a dollar sign `th:something="${expression}"` will be evaluated
- So the simplest use of Thymeleaf will have the name of a model element `th:value="${date}"`
- If the model element is an object, we can access instance variables using the dot operator `${table.number}`
- Syntax such as foreach can be used to name local variables:

```
<option th:each="table : ${tables}"
th:value="${table.id}"
th:text="'Table ' + ${table.number}">
```

- To have Thymeleaf process data in you Javascript, you need to add an attribute to the tag. E.g. `<script th:inline="javascript">`
- Then objects can be included in your JS code
- E.g. `const bookings = [[${bookings}]]` and `const tableList = [[${tables}]]`

```
1  /*<![CDATA[*/
2  let tab = document.getElementById("changeFormTable");
3  tab.setAttribute('value', /*[[${error?.tableNumber}]]*/ '1');
4  /*]]>*/
```

- If we use a value in the template, then it must be supplied in the model
- This task is completed by the Controller object
- We use the method `addAttribute`
  - E.g. `model.addAttribute("date", date.toString());`
- If the attribute is an object, it is converted into a JSON representation

```
1  private void fillModel(LocalDate date, Model model){
2    List<Booking> bookings  = restaurantService.getBookings(
     ↪  date);
3    List<Table> tables = restaurantService.getTables();
4    model.addAttribute("date", date.toString());
5    model.addAttribute("yesterday", date.minusDays(1));
6    model.addAttribute("tomorrow", date.plusDays(1));
7    model.addAttribute("bookings", bookings);
8    model.addAttribute("tables", tables);
9  }
```