Rerervation REST service Reference Documentation.
Alex Nosko – version 1.0 Release, 2018-04-28

Table of Contents

1. Preface

Project "hall-reservation" represents REST service for the reservation data interation. The project structure is based on the Spring Framework as development platform and Java Persistence API as data interation module. Aforementioned layers are combined into the module in accordance with the Spring MVC web project specification.

The project structure consists of 3 layers:

1. Web Layer.
2. Service Layer.
3. Data Access Layer.

The goal of MVC structure is to separate classes responsibilities and to make layers independent and easily extendable.

The following diagram represents the actual project layered structure.

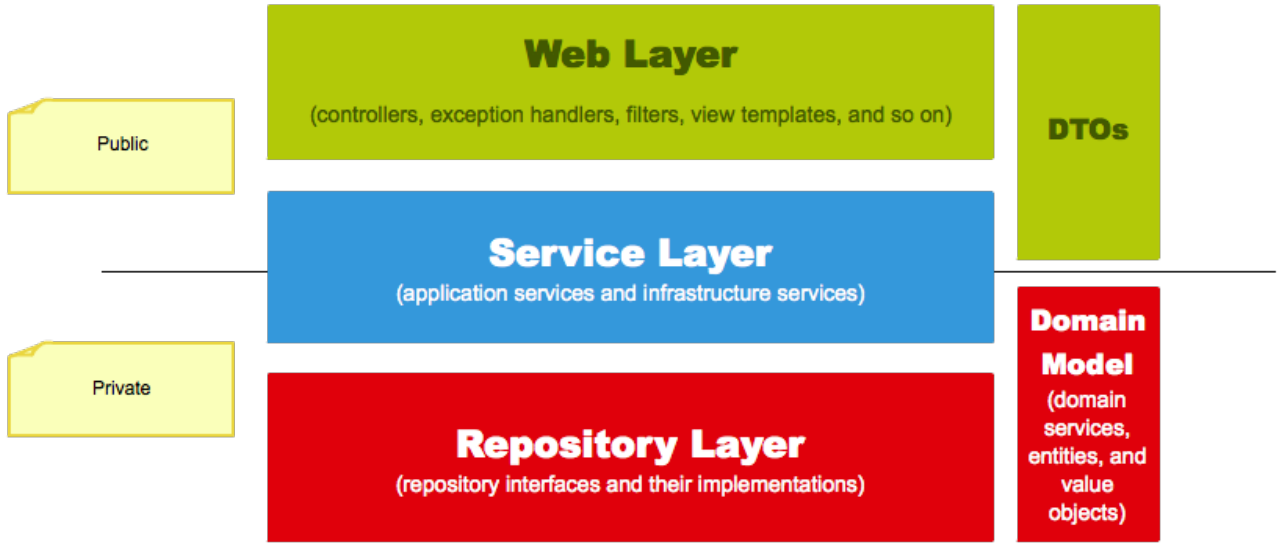


Figure 1.1

Data source is represented by the MySql database.

Following chapters desctibe each module separately and with the interaction with dependent modules.

2. Project metadata

- Version control - https://github.com/NameIess/hall-reservation

2.1 Working with controllers

Spring MVC is designed around the front controller pattern where a central `Servlet`, the `DispatcherServlet`, provides a shared algorithm for request processing while actual work is performed by configurable, delegate components. This model is flexible and supports diverse workflows.

The goal of the controller is to be an intermediary between presention layer and business logic layer.

2.1.1 Core concepts

The REST controller wokring process for request handling is represented with the following diagram.

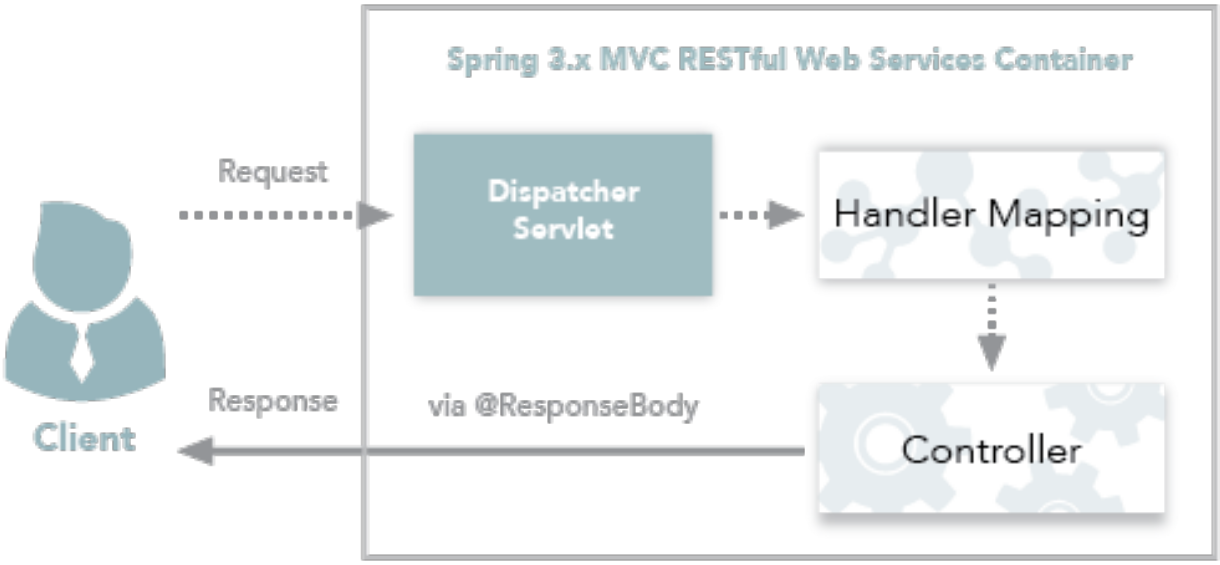


Figure 2.1.1.1

The sequence of events corresponding to the incoming HTTP request:

1. After receiving the HTTP request, `DispatcherServlet` accesses the `HandlerMapping` interface, which determines which Controller should be called, and then sends the request to the particular Controller.
2. Depending on the request type controller invokes particular business logic method and receives required data.
3. Controller creates response body object and transfer it to the client layer.

There is one REST controller in Web Layer of the "hall-reservation" project. This controller is represented by `ReservationController` class.

The following listing contains `findAllEmployees` method that works in accordance to aforementioned steps.

```
@RequestMapping(value = "/employee", method = RequestMethod.GET)
public List<Employee> findAllEmployees() {
    List<Employee> employees = employeeService.findAll();
    return employees;
}
```

Listing 2.1.1.1

Method `findAllEmployees` "listens" the `"/employee"` URI and executes in response to a request with the GET method.
After the business logic method invocation it receives data and transfers an response body object to the client.

2.1.2 Data transfer

The key difference between a traditional Spring MVC controller and the RESTful web service controller is the way the HTTP response body is created. The RESTful web service controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML.
There are two data-transfer objects that are used by `ReservationController` methods in "hall-reservation" project.
Data-transfer objects contain the complex data that separately don't satisfy with RESTful semantics of data transfer.
Method `saveReservation` receives data that are wrapped into `SaveReservationDTO` from the Client Layer.

The following listing contains `saveReservation` method source code.

```
public Reservation saveReservation(@Valid @RequestBody
SaveReservationDTO saveReservationDTO) throws ServiceException {
     Reservation savedReservation =
     reservationService.save(saveReservationDTO);

     return savedReservation;
}
```

Listing 2.1.2.1

The `@RequestBody` annotation maps the HttpRequest body to a transfer object, enabling automatic deserialization of the inbound HttpRequest body onto a Java object.

The following listing contains a `SaveReservationDTO` source code.

```
public class SaveReservationDTO {
     @NotNull
     private Long employeeId;

     @NotNull
     private Long roomId;

     @NotNull(message = "error.reservation.DateTimeFormat")
     @DateTimeFormat(pattern = "reservation.dateTimeFormat")
     private Date startTime;

     @NotNull(message = "error.reservation.DateTimeFormat")
     @DateTimeFormat(pattern = "reservation.dateTimeFormat")
     private Date endTime;
     }

// getters and setters are omitted for a space saving
```

Listing 2.1.2.2

DTO object affords to store any needed complex data encapsulate it and share.

2.1.2.1 Data validation

To prevent working with invalid data and to save the data integrity incoming data validators have been created.

Data transfer objects have particular field-level constraints that check if the field satisfies with required pattern. Validation is provided by Hibernate Validator.
The listing 2.1.2.2 contains a `SaveReservationDTO` class source code. Each field has annotations that add the particular constraint.

Current validator dictates following rules:

1. Doesn't allow `Null` values for any field.
2. All date-fields must satisfy with unify pattern.

In case of invalid data the Hibernate Validator throws an exception with a pre-defined message inside a annotation constraint.
The listing 2.1.2.1 contains a `saveReservation` method source code. This method receives the `SaveReservationDTO` instance and uses the `@Valid` annotation on an object as an indication to the validation framework to process the annotated object.

2.1.3 Defining service dependencies

In accordance to the MVC specification, Web Layer is the intermediary between a View layer and a Service Layer. `ReservationController` interacts with the Service Layer using instances associated with the help of the composition and created by the Spring platform.

The following diagram contains the ReservationController class with its Service Layer dependencies.

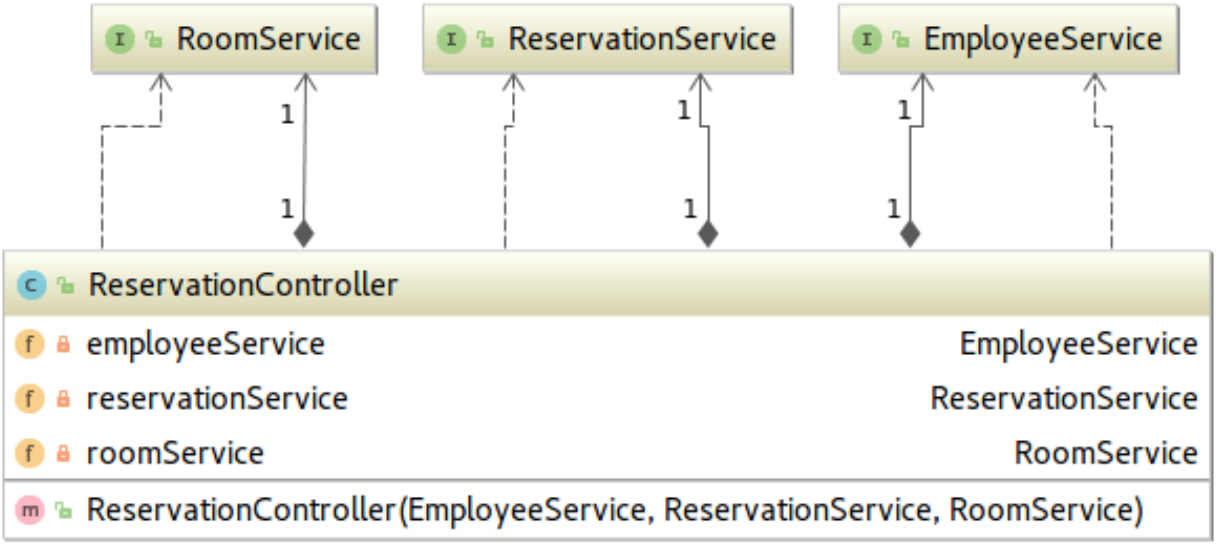


Figure 2.1.3.1

The following listing contains the `ReservationController` source code.

```
@RestController
@RequestMapping(value = "/conference")
public class ReservationController {

     private EmployeeService employeeService;
     private ReservationService reservationService;
     private RoomService roomService;

     @Autowired
     public ReservationController(
          EmployeeService employeeService,
          ReservationService reservationService,
          RoomService roomService) {
               this.employeeService = employeeService;
               this.reservationService = reservationService;
               this.roomService = roomService;
               }
     }

     @RequestMapping(value = "/employee", method =
     RequestMethod.GET)
     public List<Employee> findAllEmployees() {
          List<Employee> employees = employeeService.findAll()
          return employees;

// other methods are omitted
```

Listing 2.1.3.1

Service Layer dependencies are declared as private fields. Spring platform uses `@Autowired` annotation as an indication to create required instances and to inject dependencies by constructor parameters. After dependencies injection the `ReservationController` is ready to handle requests and invoke methods from a underlying Service Layer.

2.1.4 Exception handling

There are two an exception handling classes in a Web Layer:

1. `ServiceExceptionHandler`
2. `ValidationExceptionHandler`

`ServiceExceptionHandler` responsibility is to catch all exceptions that have been thrown from a underlying Service Layer.

The following listing contains a `ServiceExceptionHandler` class.

```
@ControllerAdvice(annotations = RestController.class)
@Order(Ordered.LOWEST_PRECEDENCE)
public class ServiceExceptionHandler extends
ResponseEntityExceptionHandler {

    @ExceptionHandler(ServiceException.class)
    public ResponseEntity<String>
    handleServiceException(ServiceException exception) {
        return new ResponseEntity<>(
            exception.getMessage(),
            HttpStatus.BAD_REQUEST);
    }
}
```

Listing 2.1.4.1

`ServiceExceptionHandler` provides a unified method of the exception handling. It prevents a code overlapping and doesn't add a new responsibility to the `ReservationController`.

Annotation with a parameter `@ControllerAdvice(annotations = RestController.class)` is used as an indication for Spring platform to add an actual advice to all controllers annotated with an `@RestController` annotation.

Annotation `@Order` is used if there are some exception handlers in a layer to set a particular order in a chain of the exception handlers.

Annotation `@ExceptionHandler(ServiceException.class)` is used to define a particular exception class, that instances will be catch by a following method. Method receives an exception, gets a exception message and wraps it with particular `HttpStatus` into `ResponseEntity`.

`ValidationExceptionHandler's` responsibility is to catch all validation exceptions that have been thrown by the Hibernate Validator.

The following listing contains a `ValidationExceptionHandler` class.

```
@ControllerAdvice(annotations = RestController.class)
@Order(Ordered.HIGHEST_PRECEDENCE)
public class ValidationExceptionHandler {

  @ExceptionHandler(MethodArgumentNotValidException.class)
  @ResponseBody
  public ResponseEntity<ErrorFormInfo>
     handleMethodArgumentNotValidException(
      MethodArgumentNotValidException exception) {

   BindingResult bindingResult = exception.getBindingResult();

   List<FieldError> fieldErrors =
bindingResult.getFieldErrors();

   ErrorFormInfo errorDto = processFieldErrors(fieldErrors);

   return new ResponseEntity<>(errorDto,
HttpStatus.BAD_REQUEST);
}

  private ErrorFormInfo processFieldErrors(List<FieldError>
     fieldErrors) {

   ErrorFormInfo dto = new ErrorFormInfo();

   for (FieldError fieldError : fieldErrors) {
      String message = fieldError.getDefaultMessage();
      String field = fieldError.getField();
      dto.addFieldError(field, message);
   }
    return dto;
  }

}
```

Listing 2.1.4.2

Method receives an exception, gets a list of invalid fields, invokes a `processFieldErrors` method, that receives the list and wraps a field name and a exception message into `ErrorFormInfo` instance.
After all composite DTO objects and a `HttpStatus` are wrapped into `ResponseEntity`.

2.1.4.1 Exception data transfer

Exception data transfer classes were created to separately store the source of the error and the error message. It helps to encapsulate an error information and it makes a response more detailed.

That classes are:

1. `FieldErrorDTO`
2. `ErrorFormInfo`

The following listing contains a `FieldErrorDTO` class source code.

```
public class FieldErrorDTO {
    private String field;
    private String message;

    // getters and setters are omitted for space saving
}
```

Listing 2.1.4.1.1

The aforementioned class contains two fields, they store an information about a field, which hasn't passed validation and a pre-defined message, which describes required rules the field must satisfy with.

As the `FieldErrorDTO` class contains an information about a single field error, to store an information about all validation exceptions `ErrorFormInfo` class was created.

The following listing contains a `ErrorFormInfo` class source code.

```
public class ErrorFormInfo {
   private List<FieldErrorDTO> fieldErrors = new ArrayList<>();

   public void addFieldError(String field, String message) {
      FieldErrorDTO error = new FieldErrorDTO(field, message);
      fieldErrors.add(error);
   }
}
```

Listing 2.1.4.1.2

`ErrorFormInfo` class is a container that stores a list of all `FieldErrorDTO` classes. Method `addFieldError` receives an error

information, wraps it into an `FieldErrorDTO` object and adds this object to the `fieldErrors` list. Refer the Listing 2.1.4.2 to see how exception data transfer objects are used within exception handlers.

2.2 Working with services

Service Layer provides a project business logic. The goal of the Service Layer is to be an intermediary between the Web Layer and the DAO Layer. This model is flexible and supports diverse workflows. In general the Service Layer contains business logic interfaces for interaction with each domain model objects.

2.2.1 Business logic

The actual project business logic is built around a "Reservation system". Domain Models are `Employee`, `Reservation`, `Room`. So the the main interfaces are represented by following classes that provide required operations for each Domain Model:

1. `EmployeeService`
2. `ReservationService`
3. `RoomService`

There is a Service Layer data validator `DateValidator` class. This class validates business data in a more specific way for a specific implementation.
The following chapters describe aforementioned classes.

2.2.1.1 Data operations

`EmployeeService` and `RoomService` classes have the same implementation and provide the same functional. These methods are:

1. `findAll()`
2. `save()`
3. `findById()`

The method name briefly explains an operation it does. As there are no an additional activity required, aforementioned methods have the same structure: they delegate execution by an invocation a particular method from a needed repository instance.

The following listing contains `save` methods from `EmployeeService` and `RoomService` implementation classes.
The main methods difference is the domain instance, that the actual service interface was developed for.

```
public Employee save(Employee employee) {
     Employee saveEmployee = employeeRepository.save(employee);
     return saveEmployee;
}
public Room save(Room room) {
     Room savedRoom = roomRepository.save(room);
     return savedRoom;
}
```

Listing 2.2.1.1.1

`ReservationService` interface is not so trivial as `EmployeeService` and `RoomService`. The first difference – data transfer objects, that are use as method parameters instead of domain `Reservation` model. Data transfer objects are light-weight objects that store only business logic required data.

The following listing contains `SaveReservationDTO` class that instance is used as method parameter of the `save` method in `ReservationService` implementation class.

```
public class SaveReservationDTO {
     @NotNull
     private Long employeeId;

     @NotNull
     private Long roomId;

     @NotNull(message = "error.reservation.DateTimeFormat")
     @DateTimeFormat(pattern = "reservation.dateTimeFormat")
     private Date startTime;

     @NotNull(message = "error.reservation.DateTimeFormat")
     @DateTimeFormat(pattern = "reservation.dateTimeFormat")
     private Date endTime;

     // getters and setters are omitted
```

Listing 2.2.1.1.2

The actual class contains domain models identifiers instead of the whole objects. The current DTO objects is also validated by the Hibernate Validator. Refer 2.1.2.1 Data validation chapter for more information.

The following listing contains a `save` method that belongs to `ReservationService` implementation class.

```
public Reservation save(SaveReservationDTO saveReservationDTO)
                                        throws ServiceException {
    // DTO patsing
    Long employeeId = saveReservationDTO.getEmployeeId();
    Employee employee = employeeService.findById(employeeId);
    Long roomId = saveReservationDTO.getRoomId();
    Room room = roomService.findById(roomId);
    Date startTime = saveReservationDTO.getStartTime();
    Date endTime = saveReservationDTO.getEndTime();

    // Data validation
    boolean isDatesValid = dateValidator.isValid(startTime,
    endTime);

    if (!isDatesValid) {
         throw new ServiceException("Dates are not valid");
    }
    // Duplicates search
    List<Reservation> reservationList = reservationRepository.
    findAllInStartTimeAndEndTimeRange(room, startTime,endTime);

    boolean isReservationAvailable = reservationList.isEmpty();

    if (!isReservationAvailable) {
         throw new ServiceException("Dates are not valid");
    }

    // Domain model creation
    Reservation reservation = reservationFactory
                  .create(employee, room, startTime, endTime);

    return savedReservation;
}
```

Listing 2.2.1.1.3

The aforementioned method executes in accordance to the following steps:

1. Parses the DTO object and obtains required fields data.
2. Uses the `DateValidator` to verify incoming data.
3. Invokes `findAllInStartTimeAndEndTimeRange` method to find reservation are already reserved.
4. Creates a domain model ready to be saved.

If the actual data don't satisfy with step 2 or with step 3, the method execution will be interrupted by a `ServiceException` throwing.

2.2.1.2 Data validation

Depending on the particular task data must satisfy with different requirements. So the `DateValidator` class was developed to verify the data to compliance with the requirements of the domain.

The following listing contains the `isValid` method that is used to validate DTO instances.

```
public boolean isValid(Date start, Date end) {
     boolean isValid = false;

     DateTime startTime = new DateTime(start);
     DateTime endTime = new DateTime(end);

     if (startTime.isBefore(endTime) &&
          !endTime.isEqual(startTime)) {
          isValid = isWorkTimePeriod(startTime, endTime);

     }

return isValid;
```

Listing 2.2.1.2.1

The aforementioned method converts data into `DateTime` instances to get more rich set of methods to operate with dates.
The first step is to check if the startTime less than `endTime`. If the data pass that check – an validation procedure execution will be delegated to `isWorkTimePeriod` method.

The following listing contains the `isWorkTimePeriod` method that is the second step of the date validation in the `DateValidator` class.

```
private static final int FRIDAY = 5;

private boolean isWorkTimePeriod(DateTime startTime, DateTime
endTime) {
     boolean isValid = false;

     int daysPeriod = Days.daysBetween(startTime,endTime)
          .getDays();
     int startDayOfWeek = startTime.getDayOfWeek();

     if (startDayOfWeek + daysPeriod <= FRIDAY) {
          boolean isStartTimeValid = isWorkTimeHour(startTime);
          boolean isEndTimeValid = isWorkTimeHour(endTime);
     }

     isValid = isStartTimeValid && isEndTimeValid;

     return isValid;
}
```

Listing 2.2.1.2.2

The aforementioned method checks if the startTime and `endTime` are constitute a period of working days. E.g. the reservation period can be from Monday till Friday, but not from Friday till Monday, because weekend is included.

This is achieved by summing two values:
`daysPeriod` and `startDayOfWeek`. The `daysPeriod` value contains information about a days range in between the `startTime` and the `endTime` that is provided with method `daysBetween` invocation. The `startDayOfWeek` value contains information about a `startTime` day number.

So if to sum that values and compare the actual result to the Friday's day number the comparison result will show if the period from the `startTime` till the `endTime` contains only workdays. If the data pass that check – an validation procedure execution will be delegated to `isWorkTimeHour` method.

The following listing contains the `isWorkTimeHour` method that is the third step of the date validation in the `DateValidator` class.

```
private static final int FRIDAY = 5;
private static final int WORKDAY_START_HOUR = 10;
private static final int WORKDAY_END_HOUR = 18;

private boolean isWorkTimeHour(DateTime dateTime) {
    boolean isValid = true;

    int hourOfDay = dateTime.getHourOfDay();
    int minuteOfHour = dateTime.getMinuteOfHour();

    if (hourOfDay < WORKDAY_START_HOUR ||
        hourOfDay > WORKDAY_END_HOUR ||
        (hourOfDay == WORKDAY_END_HOUR &&
        minuteOfHour > 0)) {
            isValid = false;
    }

    return isValid;
}
```

Listing 2.2.1.2.3

The aforementioned method checks the each date separately. If the date's time is between `WORKDAY_START_HOUR` and `WORKDAY_END_HOUR` – the date is valid. The `isWorkTimePeriod` method invokes `isWorkTimeHour` method for the `startTime` date and the `endTime` date. After all `isWorkTimePeriod` returns the combined result up to the `isValid` – invoker.

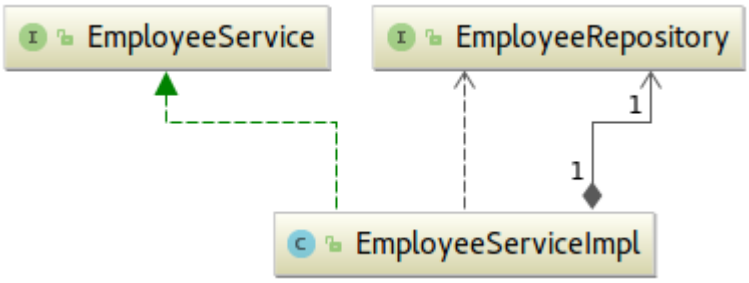
Some data validity depends on the data already stored and available in database. To prevent an duplicated values and a integrity violation appearance, there was a special database query created. This is a `findAllInStartTimeAndEndTimeRange` method that obtains `room`, `startTime`, `endTime` parameters and returns a list of reservations if parameters satisfy with already stored data. Refer the 2.4 chapter to see more information.

So the aforementioned principles check the data in different ways and provide the operation safety, fault tolerance and reliability.

2.2.2 Defining repository dependencies

In accordance to the layered architecture, Service Layer is the intermediary between a Web Layer and a Repository Layer. Domain model services implementations interact with the Repository Layer using instances associated with the help of the composition and created by the Spring platform.

The following diagram contains the EmployeeServiceImpl class with its DAO Layer dependencies.



Fragment 2.2.2.1

The following listing contains a `EmployeeServiceImpl` source code.

```
public class EmployeeServiceImpl implements EmployeeService {

     private EmployeeRepository employeeRepository;

     @Autowired
     public EmployeeServiceImpl(EmployeeRepository
     employeeRepository) {
          this.employeeRepository = employeeRepository;
     }

// other methods are omitted
```

Listing 1.15

All domain model services have the same mechanism of dependencies injection. Repository Layer dependencies are declared as private fields. Spring platform uses `@Autowired` annotation as an indication to create required instances and to inject dependencies by constructor parameters.

After dependencies injection the `EmployeeServiceImpl` is ready to invoke methods from a underlying Repository Layer.

2.2.3 Exception handling

The Service Layer has it's own mechanism of the exception handling. There are two main exception handlers in actual project Refer 2.1.4 Exception handling to see more information.
So the exception handling is not a Service Layer responsibility.

If there is an exception which has been thrown during an method execution – Service Layer methods catch the exception was thrown, wrap it and throw the new `ServiceException`, that contains an information about the exception was catched. In the other case, if there is business logic exception – Service Layer methods throw the new `ServiceException` that contains an information about the error. Regardless of the source of the error – the `ServiceException` will be thrown on the overlying layer.

The following listing contains a `findAllByEmployeeAndDateRange` method fragment, that demonstrates aforementioned principles if there is an business logic exception.

```
if (dateValidator.isValid(startTime, endTime)) {

  List<Reservation> employeeReservations =
      reservationRepository
     .findAllByStartTimeGreaterThanEqualAndEndTimeLessThanEqual
      AndEmployee(startTime, endTime, employee);

  return employeeReservations;
} else {
  throw new ServiceException("Dates are not valid");
}
```

Listing 1.15

2.2.4 Logging

In order not to lose information about changes in data a shared aspect class `ServiceLogger` was developed to provide unified advice methods for all Service Layer classes to write log files about their methods execution. Advices use shared pattern `(*com.training.playgendary.reservation.service.impl.*ServiceImpl.*(..))` to find all service classes and all methods that belong them.

The aforementioned methods are:

1. `logBeforeExecution`
2. `logAfterReturning`
3. `logAfterThrowing`

The following listings will not contain the whole pattern that is userd to find all classes and methods. This pattern will be replaced by the "PATTERN" word for a space saving.

The following listing contains `logBeforeExecution` method.

```
@Before("execution(PATTERN)")
public void logBeforeExecution(JoinPoint joinPoint) throws
Throwable {

   Log.info("Method has been called : " + joinPoint
                                          .getSignature()
                                          .getName());

   Log.info("Transferred parameters : " + Arrays.toString(
                                          joinPoint
                                          .getArgs()));
```

Listing 2.2.4.1

An annotation `@Before` indicates to execute `logBeforeExecution` method before the methods that found by PATTERN.

The following listing contains `logAfterReturning` method.

```
@@AfterReturning(pointcut = "PATTERN", returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object
result) throws Throwable {

   Log.info("Executed method : " + joinPoint
                                   .getSignature()
                                   .getName());

   Log.info("Transferred parameters : " + Arrays.toString(
                                  joinPoint
                                  .getArgs()));

   Log.info("Method returned value is : " + result);

}
```

Listing 2.2.4.2

An annotation `@AfterReturning` indicates to execute `logAfterReturning` method after a method execution where the method name matches the PATTERN.

The following listing contains `logAfterThrowing` method

```
@AfterThrowing(pointcut = "execution(PATTERN)", throwing =
"error")
public void logAfterThrowing(JoinPoint joinPoint, Throwable
error) {

   Log.error("Exception during method execution!");

   Log.error("Exception has been thrown within method : " +
         joinPoint
               .getSignature()
               .getName());
   Log.error("Exception name : " + error);
   Log.error("Exception message : " + error.getMessage());
}
```

Listing 1.18

An annotation `@AfterThrowing(pointcut = "PATTERN", throwing = "error")` indicates to execute `logAfterThrowing` method after a method throwing an exception where the method name matches the PATTERN.

So, aforementioned advices provide unified and flexible logging mechanism. The place of writing is separated from the place of execution.

2.3 Working with repositories

The goal of the DAO Layer is to provide a centralized data access and an unified interface for interaction with the data source without high coupling to data source implementation details.

2.3.1 Models mapping

The actual project data operations are built around a "conference" MySql database. There are 3 tables in there:

1. `Employee`
2. `Reservation`
3. `Room`

The following figure contains a EER-Diagram of the "reservation" database.

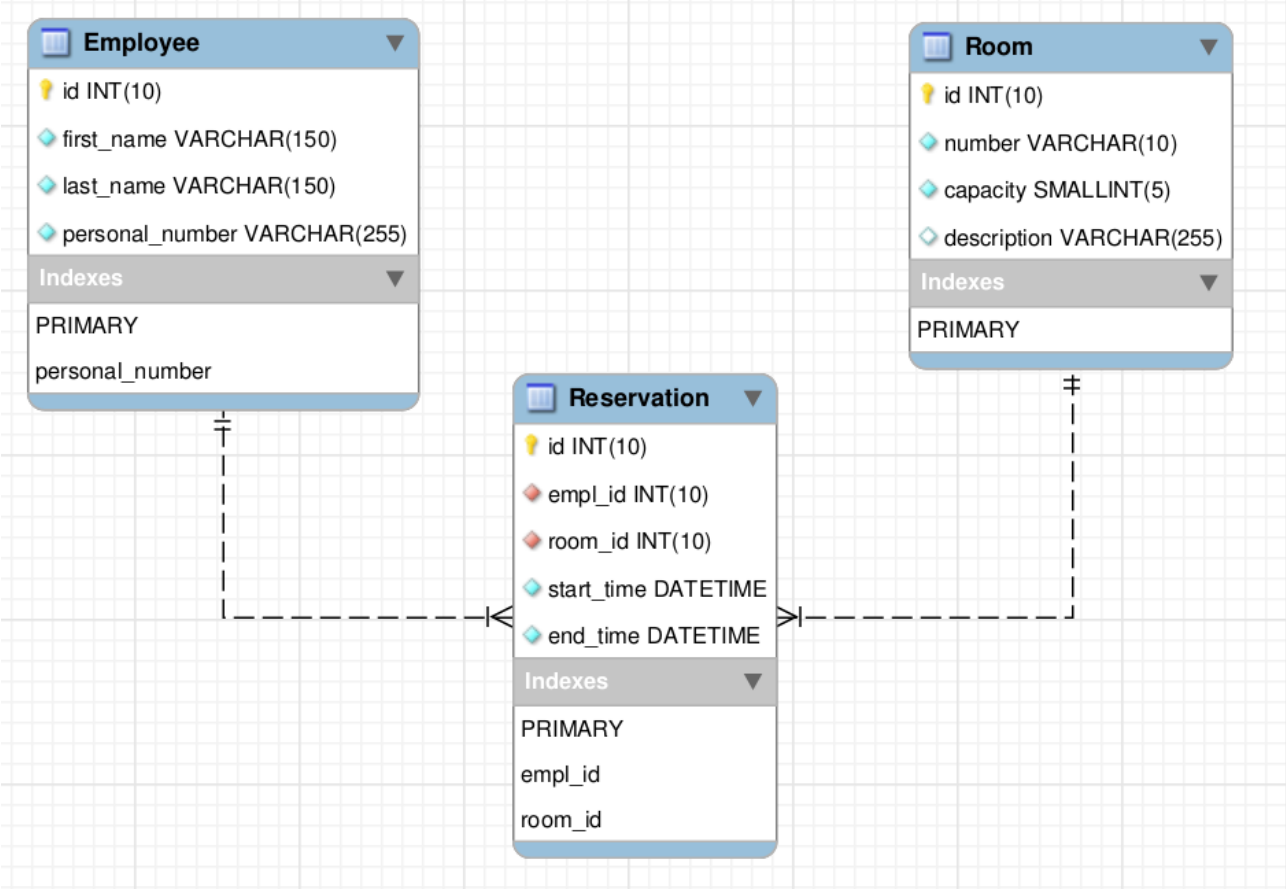


Figure 2.3.1.1

This figure shows relations between tables ant their content. Refer https://github.com/NameIess/hall-reservation/blob/master/src/main/resources/db/dump/EER-Diagram.mwb to see the source EER-Diagram file. Aforementioned tables are represented as Java classes, that satisfy the Java Bean specification. In a DAO Layer context they are named as Domain Models.

The following listing contains the `Reservation` model.

```
@Entity
@Table(name = "Reservation")
public class Reservation implements Serializable {
    private Long id;
    private Employee employee;
    private Room room;
    private Date startTime;
    private Date endTime;

    public Reservation() {
    }

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    public Long getId() {
        return id;
    }

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "empl_id")
    public Employee getEmployee() {
        return employee;
    }

    @ManyToOne(fetch = FetchType.LAZY, cascade =
{CascadeType.ALL})
    @JoinColumn(name = "room_id")
    public Room getRoom() {
        return room;
    }

    @Column(name = "start_time")
    public Date getStartTime() {
        return startTime;
    }

    @Column(name = "end_time")
    public Date getEndTime() {
        return endTime;

     // other methods were ommited for space saving
}
```

Listing 2.3.1.1

The class is marked with an `@Entity` and `@Table` annotations to inform the Java Persistence API that actual class is a Domain Model. Table columns are associated with class fields with the help of the `@Column`. Table relations are associated with dependent Domain Models with the help of the composition and appropriate annotations e.g. `@ManyToMany`, `@JoinColumn`.

2.3.2 Basic operations

The operations with data source required on the Service Layer are represented by following classes at the DAO Layer for each Domain Model:

1. `EmployeeRepository`
2. `RoomRepository`
3. `ReservationRepository`

The following listing contains the `EmployeeRepository` class.

```
@Repository
public interface EmployeeRepository extends
JpaRepository<Employee, Long> {

}
```

Listing 2.3.2.1

The aforementioned class is an interface, that extends `JpaRepository<Employee, Long>` interface parametrized with Domain Model `Employee` the interface is developed for and the id type `Long` the Domain Model is identified by.

There are no abstract methods declared in the actual repository as the business logic requirements for the `Employee` Domain Model consist of creation operations and selection operations that are already provided by the `JpaRepository`. As it was mentioned in the 2.2.1.1 Data operations chapter `Employee` and `Room` Domain Models have the same functional so the Repositories are also the same.

The following listing contains the `ReservaionRepository` class.

```
@Repository
public interface ReservationRepository extends
JpaRepository<Reservation, Long> {

List<Reservation> findAllByStartTimeGreaterThanEqual
                  AndEndTimeLessThanEqualAndEmployee(
                       Date startTime,
                       Date endTime,
                       Employee employee
                  );


@Query("SELECT r " +
     "FROM Reservation r " +
     "WHERE r.room = :room " +
"AND (r.startTime <= :startTime AND r.endTime >= :endTime " +
"OR r.startTime > :startTime AND r.startTime < :endTime " +
"OR r.endTime > :startTime AND r.endTime < :endTime) "
)
List<Reservation> findAllInStartTimeAndEndTimeRange(
                       @Param("room") Room room,
                       @Param("startTime") Date startTime,
                       @Param("endTime") Date endTime
                  );
}
```

Listing 2.3.2.2

Aforementioned class contains two abstract methods:

1. `findAllByStartTimeGreaterThanEqualAndEndTimeLessThanEqualAndEmployee`
2. `findAllInStartTimeAndEndTimeRange`

The first method is created using Spring Data JPA. Method name is based on `Reservation` Domain Model metadata and particular SQL actions. This method selects data from the `Reservation` table in between `startTime` and `endTime` including borders for the particular `Employee`.

The second method is marked with the `@Query` annotation.
The `@Query` annotation is used to create query by using the JPA query language and to bind this query directly to the method was marked by `@Query`.

This method selects data from the `Reservation` table that satisfy two of the following requirements:

1. The reservation room must be equal the `room` parameter.
2. The reservation `start_time` and the `end_time` must satisfy one of the following requirements:
   2.1. The `startTime` and the `endTime` must be into reservation `start_time` and the `end_time` range including borders.
   2.2. The reservation `start_time` must be in between the `startTime` and the `endTime`.
   2.3. The reservation `end_time` must be in between the `startTime` and the `endTime`.

As it was mentioned in the 2.2.1.2 Data validation chapter the `findAllInStartTimeAndEndTimeRange` method is used in a validation proccess.

2.4 Configuration

The following chapters describe frameworks configurations that have been used in actual project.
The configurable modules are:

1. Data access.
2. Application context.
3. Web application context.

The JavaConfig configuration model has been chosen in actual project.

2.4.1 Data access

The actual project production data source is MySql database "conference". The data access configuration is represented in the `JpaConfig` class. This class contains configuration of the JPA with Hibernate provider.

2.4.1.1 JPA

The following listing contains a class `JpaConfig` that is used like data access configuration.

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(basePackages =
"com.training.playgendary.reservation")
@PropertySource("classpath:db/jpa.properties")
public class JpaConfig {

    // other methods are ommited
}
```

Listing 2.4.1.1.1

The `@Configuration` annotation informs Spring platform that actual class is configuration class.
The `@EnableTransactionManagement` enables Spring's annotation-driven transaction management capability.
The `@EnableJpaRepositories` is an annotation to enable JPA repositories. Will scan `basePackages` packages to find repositories.

The database connection is established using JDBC `com.mysql.jdbc.Driver` driver.

The following listing contains a method `dataSource` that belongs to `JpaConfig` class.

```
private Environment env;

@Bean
public DataSource dataSource() {
     DriverManagerDataSource dataSource = new
     DriverManagerDataSource();

     dataSource
        .setDriverClassName(env.getProperty("db.driver.mysql"));

     dataSource.setUrl(env.getProperty("db.url"));

     dataSource.setUsername(env.getProperty("db.username"));

     dataSource.setPassword(env.getProperty("db.password"));

     return dataSource;
}
```

Listing 2.4.1.1.2

The aforementioned method creates an data source object, sets the required properties needed to connect to database. All properties are encapsulated in the `jpa.properties` file.

The following listing contains a method `transactionManager` that belongs to `JpaConfig` class.

```
@Bean
public PlatformTransactionManager transactionManager(
                                        EntityManagerFactory emf) {
JpaTransactionManager transactionManager =
                                    new JpaTransactionManager();
     transactionManager.setEntityManagerFactory(emf);

      return transactionManager;
}
```

Listing 2.4.1.1.3

The Entity Manager factory requires a transaction manager for transactional data access.

```
@Bean
public LocalContainerEntityManagerFactoryBean
entityManagerFactory() {

    HibernateJpaVendorAdapter vendorAdapter =
                                new HibernateJpaVendorAdapter();

    LocalContainerEntityManagerFactoryBean em =
                  new LocalContainerEntityManagerFactoryBean();

    em.setDataSource(dataSource());

    em.setPackagesToScan(
                      "com.training.playgendary.reservation");

    em.setJpaVendorAdapter(vendorAdapter);

    em.setJpaProperties(additionalProperties());

    return em;
}
```

Listing 2.4.1.1.4

The `entityManagerFactory` method injects the aforementioned dataSource bean, creates `vendorAdapter` property that specifies the implementation of a particular JPA provider – `HibernateJpaVendorAdapter`, sets packages to scan Domain Models using `setPackagesToScan`, sets properties to JPA provider using `setJpaProperties`.

2.4.1.2 Hibernate

In actual project Hibernate has the provider role.

The following listing contains a method `additionalProperties` that receives Hibernate properties from the `jpa.properties` file.

```
public Properties additionalProperties() {
    Properties properties = new Properties();

    properties
             .setProperty("hibernate.dialect.mysql",
             env.getProperty("hibernate.dialect.mysql"));

    properties
             .setProperty("hibernate.max_fetch_depth",
             env.getProperty("hibernate.max_fetch_depth"));

    properties
             .setProperty("hibernate.jdbc.fetch_size",
             env.getProperty("hibernate.jdbc.fetch_size"));

    properties
             .setProperty("hibernate.jdbc.batch_size",
             env.getProperty("hibernate.jdbc.batch_size"));

    properties
             .setProperty("hibernate.show_sql",
             env.getProperty("hibernate.show_sql"));

    properties
             .setProperty("hibernate.format_sql",
             env.getProperty("hibernate.format_sql"));

    return properties;
}
```

Listing 2.4.1.2.1

The aforementioned properties have been set to configure the Hibernate. Refer https://github.com/NameIess/hall-reservation/blob/master/src/main/resources/db/jpa.properties to see the particular properties values.

2.4.2 Application context

The actual project application context configuration is represented by `AppConfig` class. There are methods that declare resource handlers, message converters, message source.

The following listing contains a class `JpaConfig` that is used like data access configuration.

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages =
{"com.training.playgendary.reservation"})
@EnableTransactionManagement
public class AppConfig implements WebMvcConfigurer
```

Listing 2.4.2.1

Application context is configured like an Web application with the help of the `@EnableWebMvc` annotation and implementation of the `WebMvcConfigurer` interface.

Methods in the `AppConfig` class registry resource handlers, message converters in accordance to REST API requirements, message source.

2.4.3 Deployment

The deployment configuration is represented by `WebConfig` class. This class extends `AbstractAnnotationConfigDispatcherServletInitializer` class that provides methods to configure deployment proccess.

The following listing contains a WebConfig class source code.

```
public class WebConfig extends
AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    public void onStartup(ServletContext servletContext) throws
ServletException {

//Create the root Spring application context

    AnnotationConfigWebApplicationContext rootContext =
                new AnnotationConfigWebApplicationContext();
    rootContext.register(JpaConfig.class);

    ContextLoaderListener contextLoaderListener =
                new ContextLoaderListener(rootContext);
    servletContext.addListener(contextLoaderListener);

//Create the dispatcher servlet's Spring application context

    AnnotationConfigWebApplicationContext servletAppContext =
                new AnnotationConfigWebApplicationContext();

    servletAppContext.register(AppConfig.class);

    DispatcherServlet dispatcherServlet =
                new DispatcherServlet(servletAppContext);

    dispatcherServlet.setThrowExceptionIfNoHandlerFound(true);

//Register and map the dispatcher servlet

    ServletRegistration.Dynamic dispatcher =
    servletContext.addServlet("dispatcher", dispatcherServlet);

    dispatcher.setLoadOnStartup(1);
    dispatcher.addMapping("/");

    FilterRegistration.Dynamic encodingFilter =
              servletContext.addFilter("encoding-filter", new
              CharacterEncodingFilter());

    encodingFilter.setInitParameter("encoding", "UTF-8");
    encodingFilter.setInitParameter("forceEncoding", "true");
    encodingFilter.addMappingForUrlPatterns(null, true, "/*");
}
```

Listing 2.4.3.1

The aforementioned method creates configured context instances, listeners, filters, registries dispatcher servlet and assembles them together.

2.4.4 Used dependencies

There is a dependencies list were used in project development:

1. spring-core
2. spring-context
3. spring-web
4. spring-webmvc
5. spring-jdbc
6. spring-orm
7. h2
8. mysql-connector-java
9. spring-aop
10. aspectjtools
11. aspectjweaver
12. aspectjrt
13. hibernate-core
14. hibernate-jpa-2.1-api
15. spring-data-jpa
16. hibernate-validator
17. validation-api
18. jackson-core
19. jackson-databind
20. javax.servlet-api
21. commons-lang3
22. joda-time-hibernate
23. joda-time
24. log4j-core
25. spring-test
26. junit
27. mockito-all
28. testing