1- Problem Definition

The goal is to implement two services: EmployeeService and EventService.

2- Environment

In this section the architecture and the instructions to setup the environment to run the application is described.

2-1- Architecture

Following the microservice architecture, EmployeeService and EventService are developed in two separate spring boot application. Also a separate Docker container have been used for databases and the message broker.

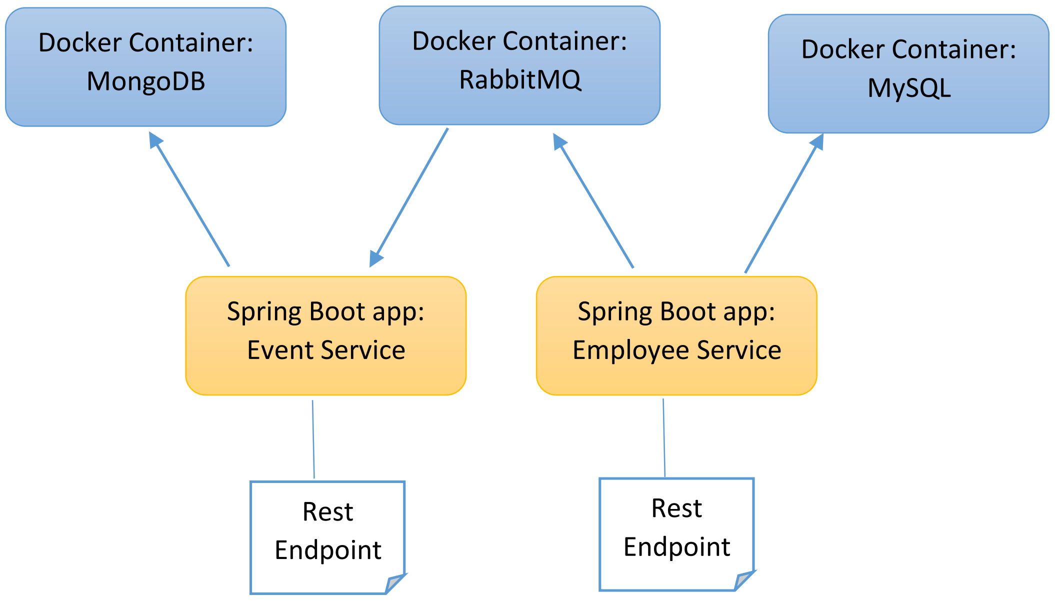

Docker Container: MongoDB
Docker Container: RabbitMQ
Docker Container: MySQL
Spring Boot app: Event Service
Spring Boot app: Employee Service
Rest Endpoint
Rest Endpoint

Figure 1- The Architecture

2-2- Installing Docker containers

The related Dockerfile for each container can be found in the "docker" folder in EmployeeService application as shown below:

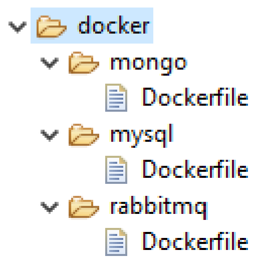


Figure 2- docker files

In order to install docker containers the following commands can be used:

For MySQL:

```
cd docker/mysql
sudo docker build . -t mysql:employee
sudo docker run -p 3306:3306 mysql:employee
```

For MongoDB:

```
cd docker/mongo
sudo docker build . -t mongo:events
sudo docker run -p 27017:27017 mongo:events
```

For RabbitMQ:

```
cd docker/rabbitmq
sudo docker build . -t rabbitmq:events
sudo docker run -p 15672:15672 -p 5672:5672 rabbitmq:events
```

Running the applications

You need to run two java applications:

- EmployeeService
- EventService

Having exported the jar files, the following commands can be used to run the applications:

EmployeeService:

```
java -jar EmployeeService.jar --server.port=8080 --
spring.datasource.url=jdbc:mysql://192.168.1.20:3306/employeeDB --
spring.rabbitmq.host=192.168.1.20 --spring.rabbitmq.port=5672
```

EventService:

```
java -jar EventService.jar  --server.port=8081 --
spring.data.mongodb.host=192.168.1.20 --spring.data.mongodb.port=27017
--spring.rabbitmq.host=192.168.1.20 --spring.rabbitmq.port=5672
```

You can change the IP and Port of databases and the message broker according to your environment.

2-3- Running from source code

To run from the source code you may want to edit application.properties files according to your environment.

Each project has two application.properties, one for production and one for tests, in the following paths:

- src/main/resources/application.properties
- src/test/resources/application.properties

3- Execution

The applications have for Employee Service and Event Service expose some REST API to read and write required data. All the API documentation is available online here on postman.

Employee Service application employs a token based authentication. Therefore, in order to call any other service, you must login before and get a token. Then the token must be sent as a header parameter in every request.

For simplicity, Event Service does not require any authentication.

3-1- Login API

As an example the login API is described here.

Endpoint: employee_service_host/api/v1/login

Credentials must be sent as a JSON body. Please don’t forget to set the media type to application/json. you can login using the default account:

```
{
   "username": "admin",
   "password": "admin"
}
```

4- Design

In this section design decisions and the main purpose of core objects is explained.

4-1- Employee Service

In the following the design details of Employee service is described.

4-1-1- Domain Objects

This layer contains the domain objects required by application logic.

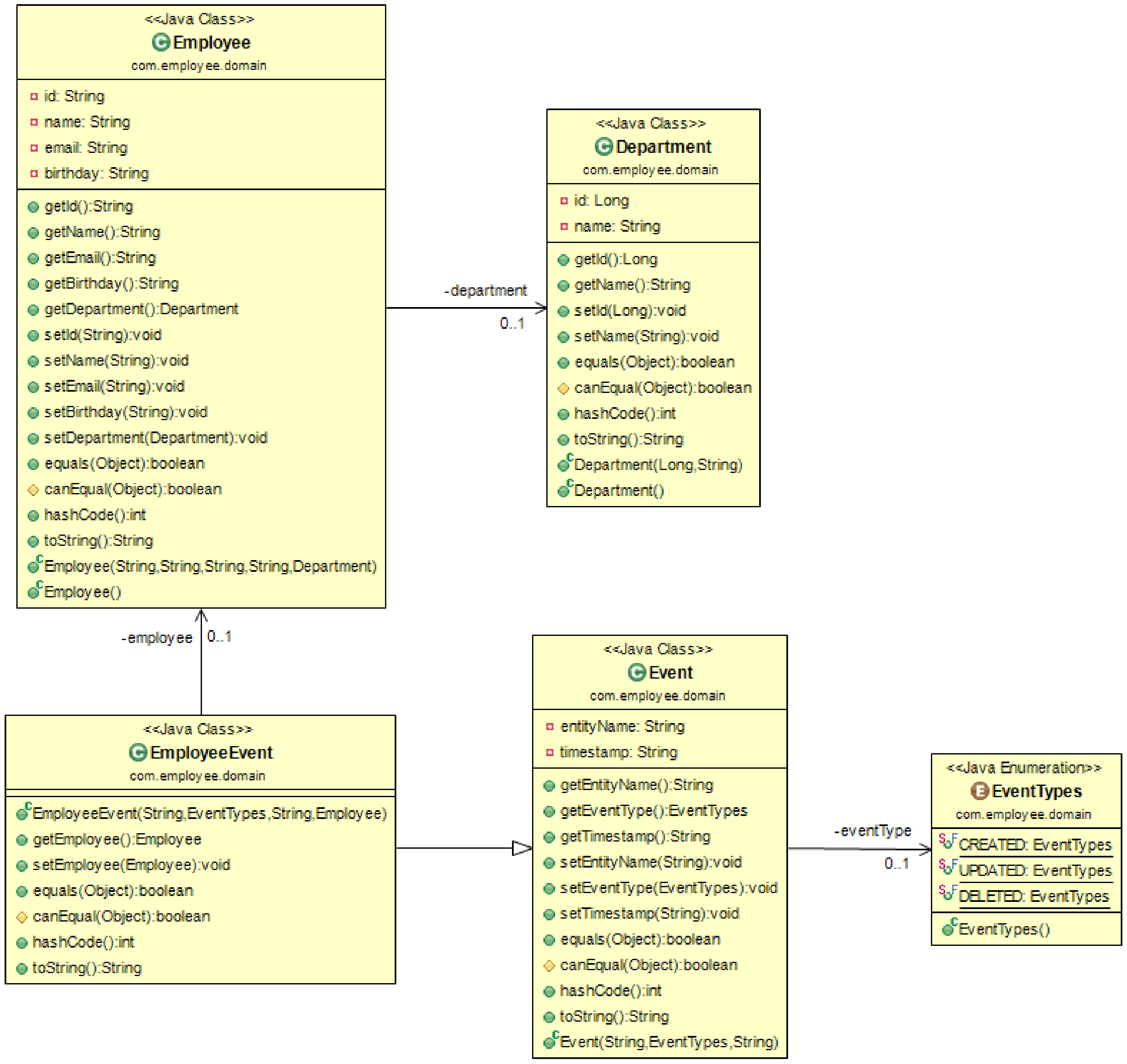
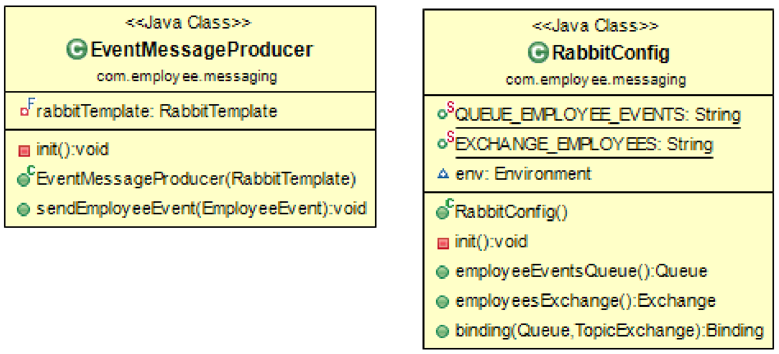


Figure 3-Domain Model

4-1-2- Messaging

The classes related to messaging can be seen here. RabbitMQ is configured via a config class and EventMessageProducer's sendEmployeeEvent is called when a change occurs. Then the event is written to a queue.

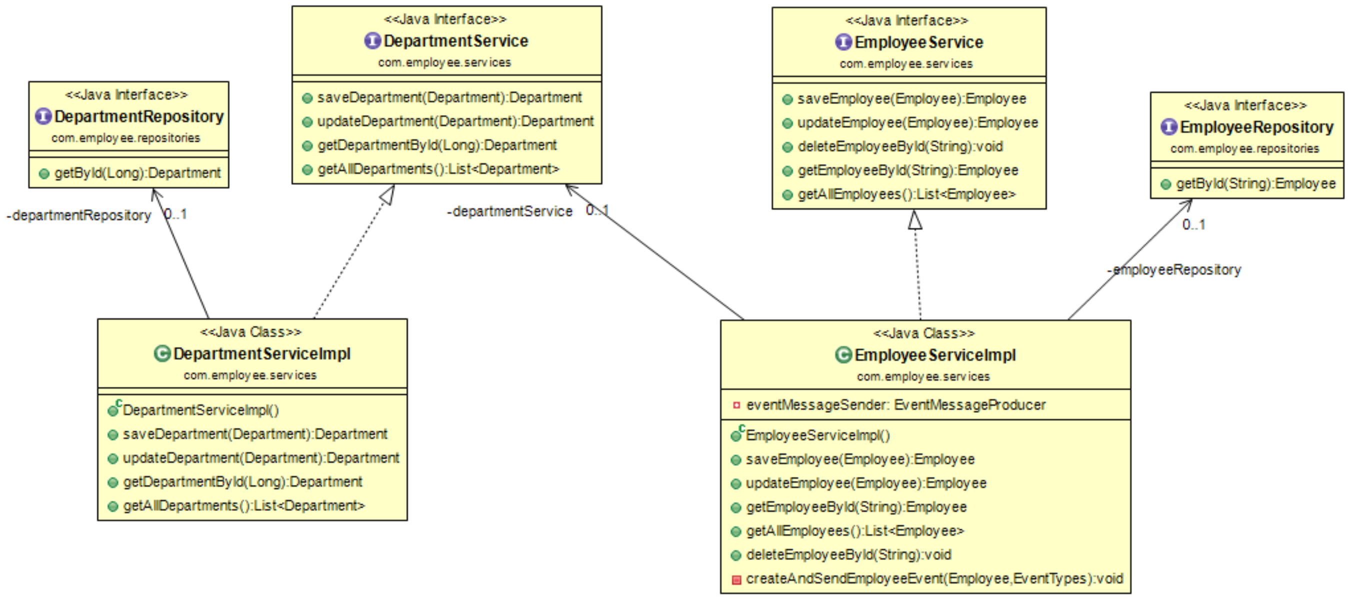


<<Java Class>>
EventMessageProducer
com.employee.messaging
rabbitTemplate: RabbitTemplate
init():void
EventMessageProducer(RabbitTemplate)
sendEmployeeEvent(EmployeeEvent):void
<<Java Class>>
RabbitConfig
com.employee.messaging
QUEUE_EMPLOYEE_EVENTS: String
EXCHANGE_EMPLOYEES: String
env: Environment
RabbitConfig()
init():void
employeeEventsQueue():Queue
employeesExchange():Exchange
binding(Queue,TopicExchange):Binding

Figure 4-Messaging

4-1-3- Services

The service layer provides the fundamental services which are available for employees and departments. There is a EmployeeServiceImpl which implements the general methods mentioned in EmployeeService and performs DAO operations via EmployeeRepository. The same story goes with DepartmenService too.



<<Java Interface>>
DepartmentRepository
com.employee.repositories
getById(Long):Department
<<Java Interface>>
DepartmentService
com.employee.services
saveDepartment(Department):Department
updateDepartment(Department):Department
getDepartmentById(Long):Department
getAllDepartments():List<Department>
<<Java Interface>>
EmployeeService
com.employee.services
saveEmployee(Employee):Employee
updateEmployee(Employee):Employee
deleteEmployeeById(String):void
getEmployeeById(String):Employee
getAllEmployees():List<Employee>
<<Java Interface>>
EmployeeRepository
com.employee.repositories
getById(String):Employee
-departmentRepository
0..1
-departmentService
0..1
0..1
-employeeRepository
<<Java Class>>
DepartmentServiceImpl
com.employee.services
DepartmentServiceImpl()
saveDepartment(Department):Department
updateDepartment(Department):Department
getDepartmentById(Long):Department
getAllDepartments():List<Department>
<<Java Class>>
EmployeeServiceImpl
com.employee.services
eventMessageSender: EventMessageProducer
EmployeeServiceImpl()
saveEmployee(Employee):Employee
updateEmployee(Employee):Employee
getEmployeeById(String):Employee
getAllEmployees():List<Employee>
deleteEmployeeById(String):void
createAndSendEmployeeEvent(Employee,EventTypes):void

Figure 5- Services

4-1-4- Controller Layer

Controller Layer which is responsible for handling REST requests is explained in this section.

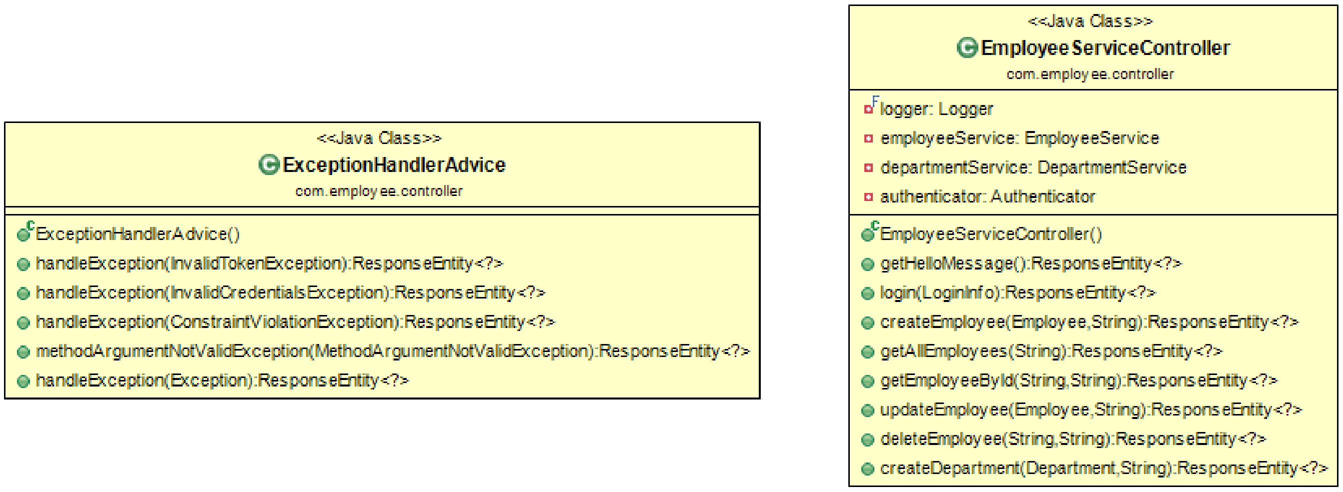

<<Java Class>>
ExceptionHandlerAdvice
com.employee.controller
ExceptionHandlerAdvice()
handleException(InvalidTokenException):ResponseEntity<?>
handleException(InvalidCredentialsException):ResponseEntity<?>
handleException(ConstraintViolationException):ResponseEntity<?>
methodArgumentNotValidException(MethodArgumentNotValidException):ResponseEntity<?>
handleException(Exception):ResponseEntity<?>
<<Java Class>>
EmployeeServiceController
com.employee.controller
logger: Logger
employeeService: EmployeeService
departmentService: DepartmentService
authenticator: Authenticator
EmployeeServiceController()
getHelloMessage():ResponseEntity<?>
login(LoginInfo):ResponseEntity<?>
createEmployee(Employee,String):ResponseEntity<?>
getAllEmployees(String):ResponseEntity<?>
getEmployeeById(String,String):ResponseEntity<?>
updateEmployee(Employee,String):ResponseEntity<?>
deleteEmployee(String,String):ResponseEntity<?>
createDepartment(Department,String):ResponseEntity<?>

Figure 6- Controller Layer

As part of the MVC pattern, the Rest Controller is designed to receive user requests, talk to the core and return a response. It currently offers the required methods for reading and writing employee data along with departments. In all services if the operation is successful a HttpStatus of OK/Created is returned along with a JSON response.

In case of exception, a separate ExceptionHandlerAdvice is designed to keep the REST Controller codes clean (thanks to SpringBoot). ExceptionHandlerAdvice returns a proper JSON response and HttpStatus according to the occurred incident.

4-1-5- Authentication

In this application a simple, from scratch token based authentication approach is taken. AuthenticatorImpl class maintains a hash map of logged in users and their corresponding tokens. When a new user logs in with correct credentials the maps get updated. In all the REST calls isTokenValid is called to check user validity.

Please note that this approach is chosen here only for simplicity. More sophisticated schemes such as JWT, Basic Auth (not really a suitable approach for production) can be employed for this purpose. Spring security has a lot to offer.

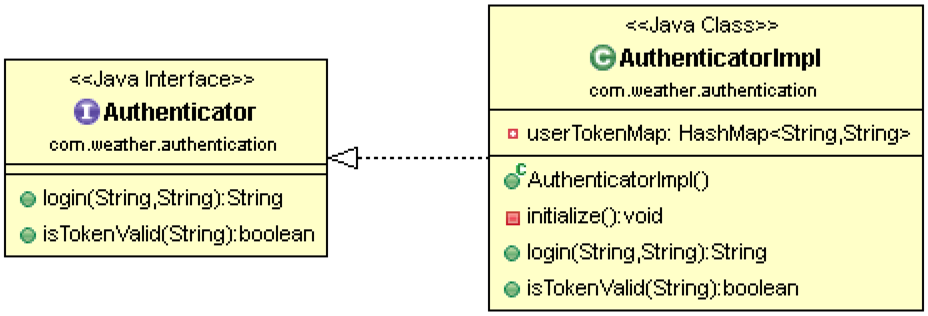

<<Java Interface>>
Authenticator
com.weather.authentication
login(String,String):String
isTokenValid(String):boolean
<<Java Class>>
AuthenticatorImpl
com.weather.authentication
userTokenMap: HashMap<String,String>
AuthenticatorImpl()
initialize():void
login(String,String):String
isTokenValid(String):boolean

Figure 7- Authentication

4-1-6- Exceptions

Custom exception classes can be seen here.

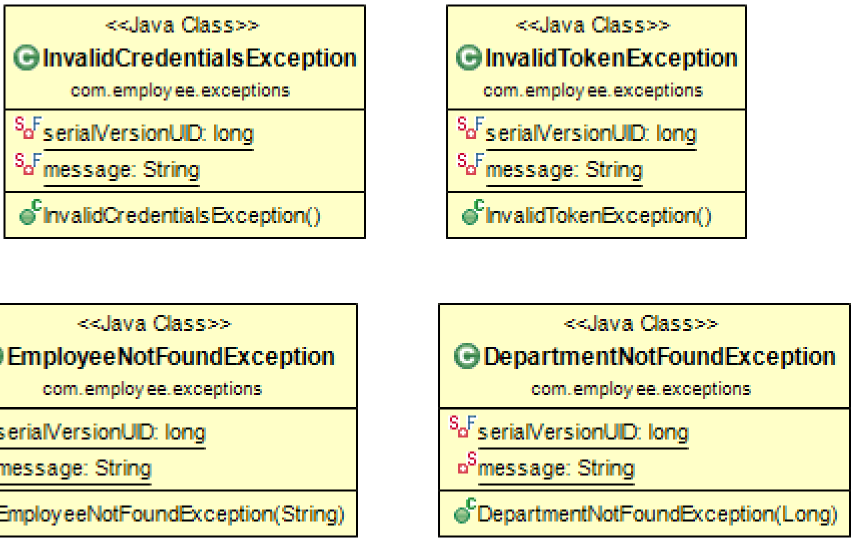


Figure 8-Custom Exception Classes

4-1-7- Tests

Unit test and integration tests are written in multiple classes. It is worth to mention that in order to separate development and production environment, an in memory database instead of main MySQL, a test collection in mongoDB, and a test queue in RabbitMQ have been used. These settings can be found in the application.properties of test directory.

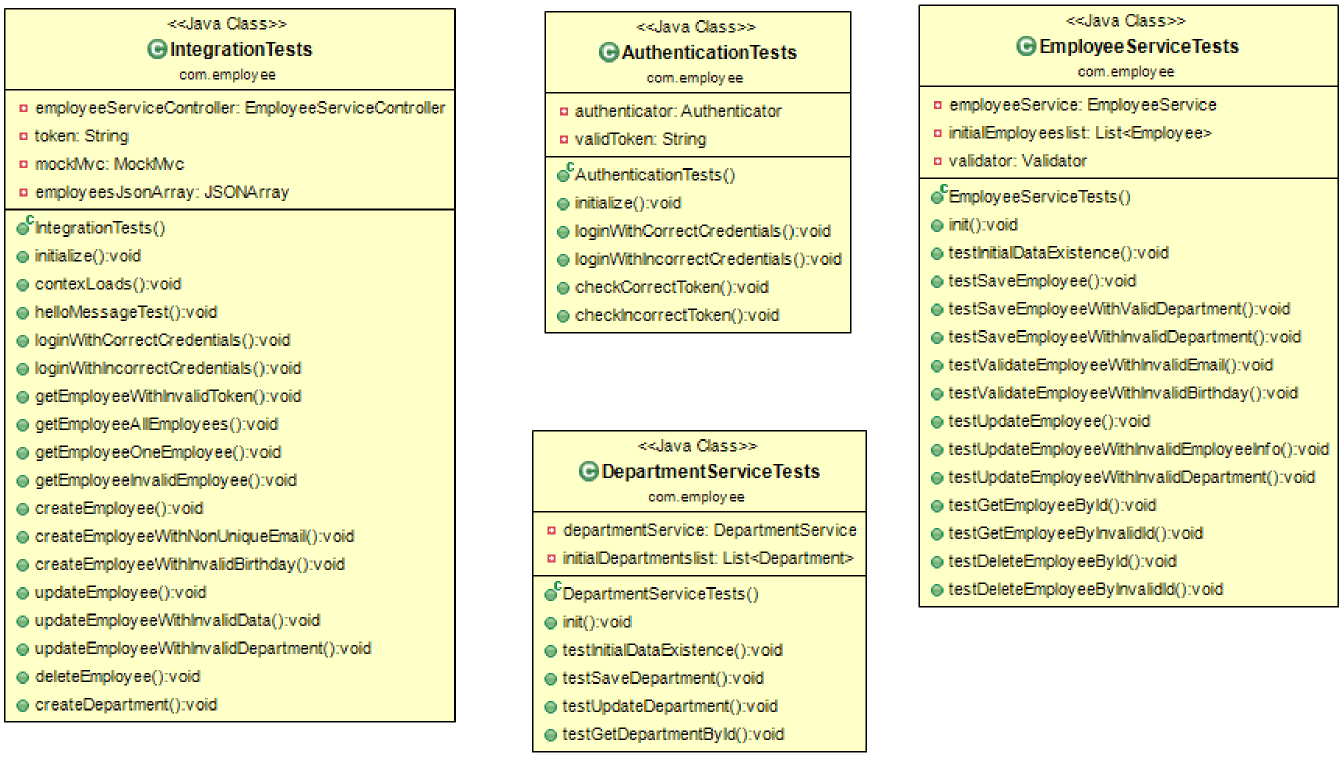
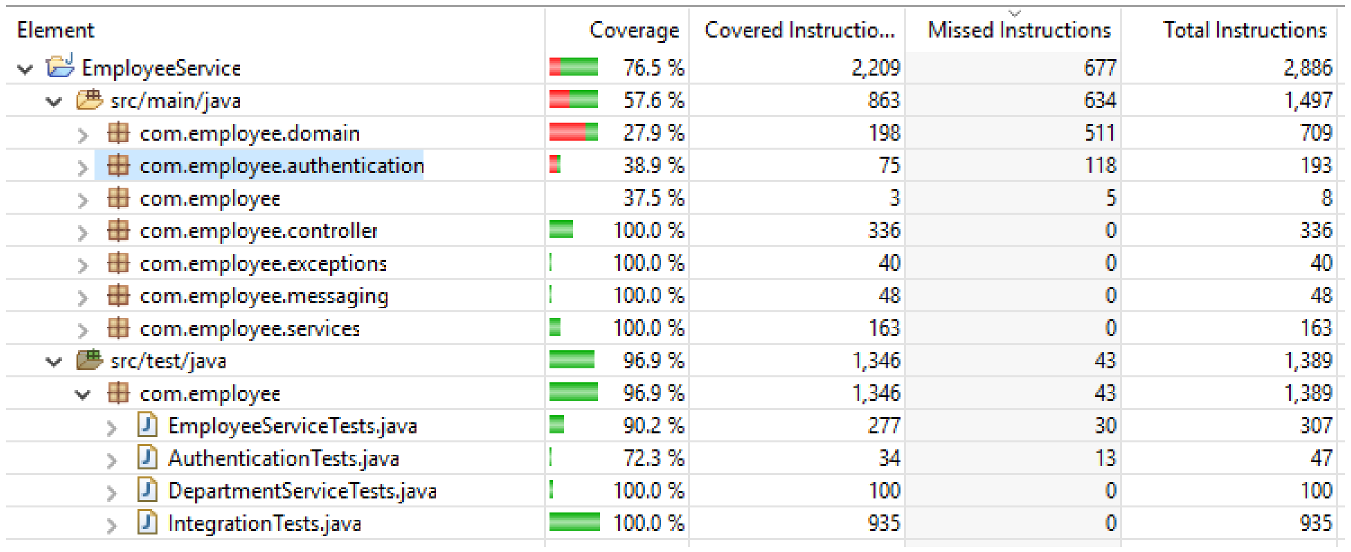

<<Java Class>>
IntegrationTests
com.employee
employeeServiceController: EmployeeServiceController
token: String
mockMvc: MockMvc
employeesJsonArray: JSONArray
IntegrationTests()
initialize():void
contextLoads():void
helloMessageTest():void
loginWithCorrectCredentials():void
loginWithIncorrectCredentials():void
getEmployeeWithInvalidToken():void
getEmployeeAllEmployees():void
getEmployeeOneEmployee():void
getEmployeeInvalidEmployee():void
createEmployee():void
createEmployeeWithNonUniqueEmail():void
createEmployeeWithInvalidBirthday():void
updateEmployee():void
updateEmployeeWithInvalidData():void
updateEmployeeWithInvalidDepartment():void
deleteEmployee():void
createDepartment():void
<<Java Class>>
AuthenticationTests
com.employee
authenticator: Authenticator
validToken: String
AuthenticationTests()
initialize():void
loginWithCorrectCredentials():void
loginWithIncorrectCredentials():void
checkCorrectToken():void
checkIncorrectToken():void
<<Java Class>>
DepartmentServiceTests
com.employee
departmentService: DepartmentService
initialDepartmentslist: List<Department>
DepartmentServiceTests()
init():void
testInitialDataExistence():void
testSaveDepartment():void
testUpdateDepartment():void
testGetDepartmentById():void
<<Java Class>>
EmployeeServiceTests
com.employee
employeeService: EmployeeService
initialEmployeeslist: List<Employee>
validator: Validator
EmployeeServiceTests()
init():void
testInitialDataExistence():void
testSaveEmployee():void
testSaveEmployeeWithValidDepartment():void
testSaveEmployeeWithInvalidDepartment():void
testValidateEmployeeWithInvalidEmail():void
testValidateEmployeeWithInvalidBirthday():void
testUpdateEmployee():void
testUpdateEmployeeWithInvalidEmployeeInfo():void
testUpdateEmployeeWithInvalidDepartment():void
testGetEmployeeById():void
testGetEmployeeByInvalidId():void
testDeleteEmployeeById():void
testDeleteEmployeeByInvalidId():void

Figure 9- Test Classes

4-1-8- Coverage

Below figure demonstrates the code coverage for the tests. Note that domain classes indicate low percent because of auto generated getters and setters.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
EmployeeService	76.5 %	2,209	677	2,886
src/main/java	57.6 %	863	634	1,497
com.employee.domain	27.9 %	198	511	709
com.employee.authentication	38.9 %	75	118	193
com.employee	37.5 %	3	5	8
com.employee.controller	100.0 %	336	0	336
com.employee.exceptions	100.0 %	40	0	40
com.employee.messaging	100.0 %	48	0	48
com.employee.services	100.0 %	163	0	163
src/test/java	96.9 %	1,346	43	1,389
com.employee	96.9 %	1,346	43	1,389
EmployeeServiceTests.java	90.2 %	277	30	307
AuthenticationTests.java	72.3 %	34	13	47
DepartmentServiceTests.java	100.0 %	100	0	100
IntegrationTests.java	100.0 %	935	0	935

Figure 10 – Coverage

4-2- Events Service

In the following the design details of Event service is described.

4-2-1- Messaging

This layer is responsible for listening to the proper queues and handle the incoming messages.

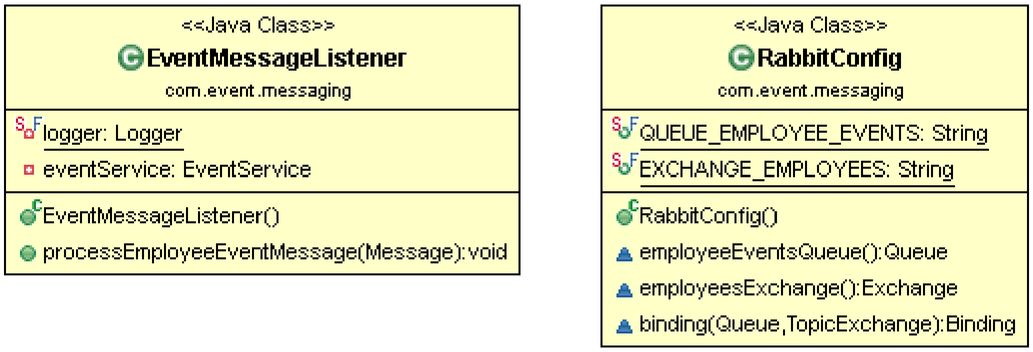

<<Java Class>>
EventMessageListener
com.event.messaging
logger: Logger
eventService: EventService
EventMessageListener()
processEmployeeEventMessage(Message):void
<<Java Class>>
RabbitConfig
com.event.messaging
QUEUE_EMPLOYEE_EVENTS: String
EXCHANGE_EMPLOYEES: String
RabbitConfig()
employeeEventsQueue():Queue
employeesExchange():Exchange
binding(Queue,TopicExchange):Binding

Figure 11-Messaging Layer

4-2-2- Events Service Layer

EventService class provides the required methods to reading and writing events from/to mongoDB. A NoSQL DB has been chosen here because the schema-free nature and writing speed can be a help when working with events.

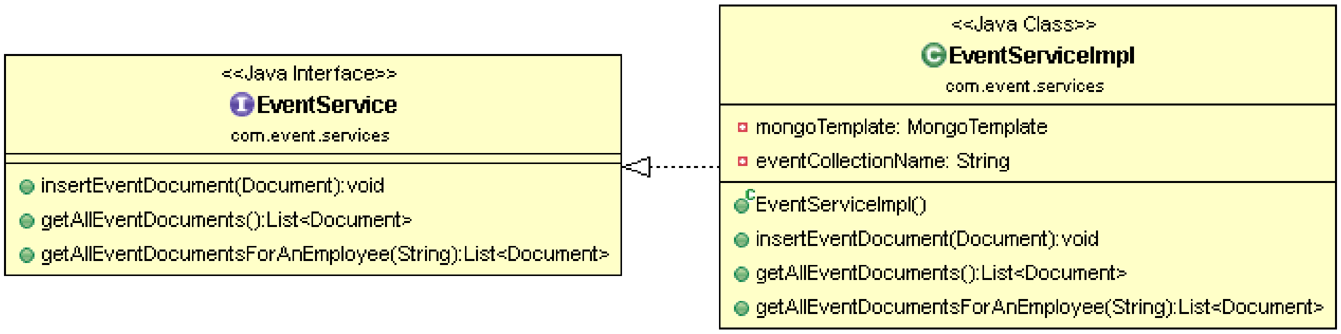

<<Java Interface>>
EventService
com.event.services
insertEventDocument(Document):void
getAllEventDocuments():List<Document>
getAllEventDocumentsForAnEmployee(String):List<Document>
<<Java Class>>
EventServiceImpl
com.event.services
mongoTemplate: MongoTemplate
eventCollectionName: String
EventServiceImpl()
insertEventDocument(Document):void
getAllEventDocuments():List<Document>
getAllEventDocumentsForAnEmployee(String):List<Document>

Figure 12-Event Service Layer

4-2-3- Controller Layer

Controller Layer which is responsible for handling REST requests is only used to read the events.

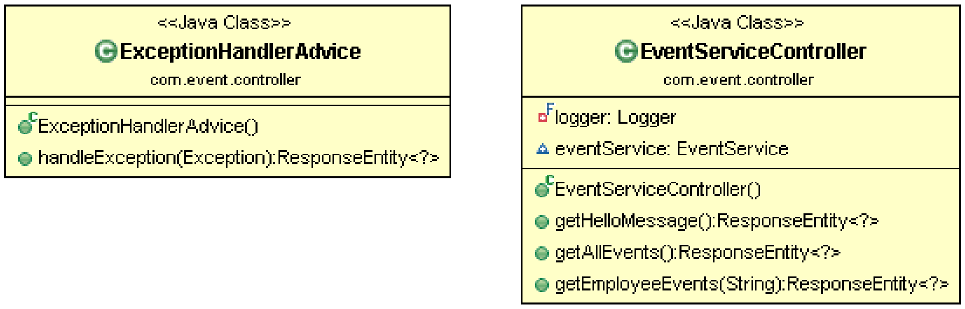

<<Java Class>>
ExceptionHandlerAdvice
com.event.controller
ExceptionHandlerAdvice()
handleException(Exception):ResponseEntity<?>
<<Java Class>>
EventServiceController
com.event.controller
logger: Logger
eventService: EventService
EventServiceController()
getHelloMessage():ResponseEntity<?>
getAllEvents():ResponseEntity<?>
getEmployeeEvents(String):ResponseEntity<?>

Figure 13- Controller Layer

4-2-4- Tests

Unit test and integration tests are written in multiple classes. It is worth to mention that in order to separate development and production environment, a test collection in mongo db. These settings can be found in the application.properties of test directory.

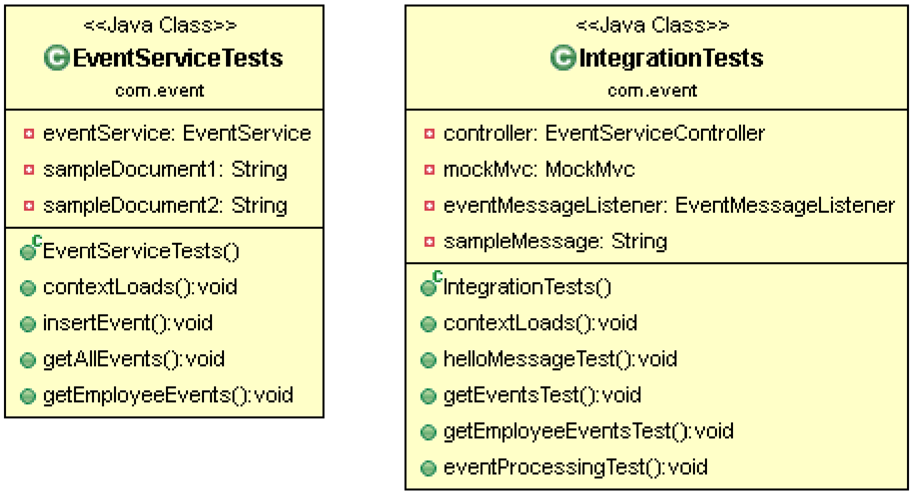
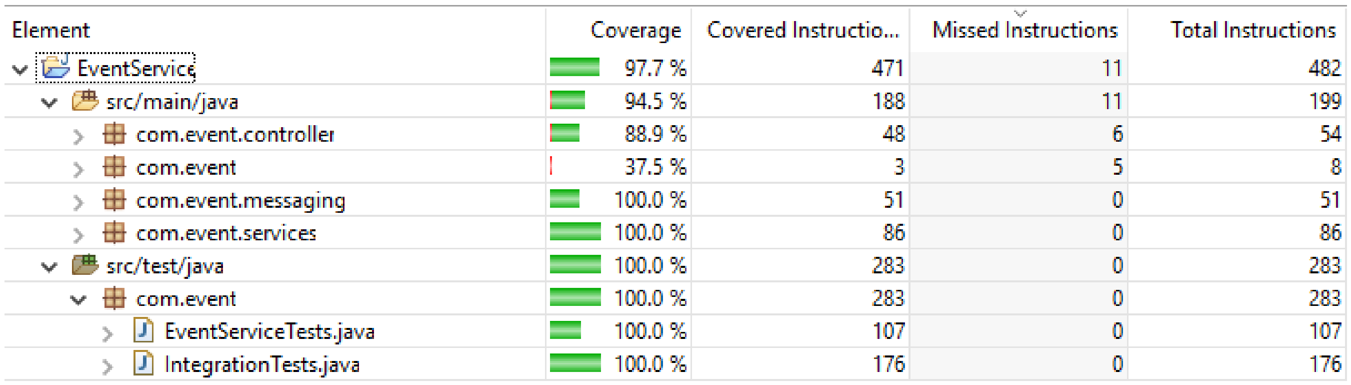


Figure 14- Test Classes

4-2-5- Coverage

Below figure demonstrates the code coverage for the tests.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
EventService	97.7 %	471	11	482
src/main/java	94.5 %	188	11	199
com.event.controller	88.9 %	48	6	54
com.event	37.5 %	3	5	8
com.event.messaging	100.0 %	51	0	51
com.event.services	100.0 %	86	0	86
src/test/java	100.0 %	283	0	283
com.event	100.0 %	283	0	283
EventServiceTests.java	100.0 %	107	0	107
IntegrationTests.java	100.0 %	176	0	176

Figure 15 - Coverage