HW1: Mid-term assignment report

Rafael kauati [105925], v2024-04-12

1 Introduction

1.1 Overview of the work

This report presents the midterm individual project required for TQS, covering both the software product features and the adopted quality assurance strategy.

This project entails the development of a REST API using the Spring Boot framework, offering functionality akin to a travel bus search and booking service, primarily inspired by FlixBus. Its primary objectives encompass:

1. Facilitating the search for bus connections (trips) between two cities.
2. Enabling the booking of reservations for passengers.

The development approach includes comprehensive testing strategies such as Unit Tests, Service-Level Tests, Web-Interface tests, and Integration Tests to ensure robustness and reliability.

Data pertaining to travel connections and user-booked tickets are stored within a MySQL database. Additionally, the application integrates with an external API to fetch real-time exchange rates for various currencies, enhancing user flexibility (EUR, USD, JPY and GBP).

Furthermore, a sleek and intuitive web interface has been crafted using React/Vite, offering a minimalist yet effective means for users to interact with the API, visualize data, and seamlessly utilize its features.

1.2 Current limitations

Generally speaking, the application (Backend + Frontend) itself is still pretty simple in terms of feature and complexity, regarding the backend application :

- The logical layer could have a more sophisticated logical verification, especially when it comes to feature of the currency conversion
- The data format of response should be more “customized”, (e.g., the data format of ticket details that is returned from the backend to the client) , currently it returns the proper data with simple format that still needs to be organized the frontend, something that should be dealt in the service side of the project, with properly data body containing messages with success/failure of operations that could be interpreted in the front-end
- The token logic also is not perfectly implemented, i.e., currently, in order to search for, e.g, all the tickets one client bought , it uses the client name (in the backend is identified as “Owner”. In a more realistic software application this token should be a proper real token, like an generated and tracked id by the backend and stored as cookie in the frontend

Now regarding the front end:

- The front end currently consists on a single page with all the components that interact with the api, which should be distributed between others components

2 Product specification

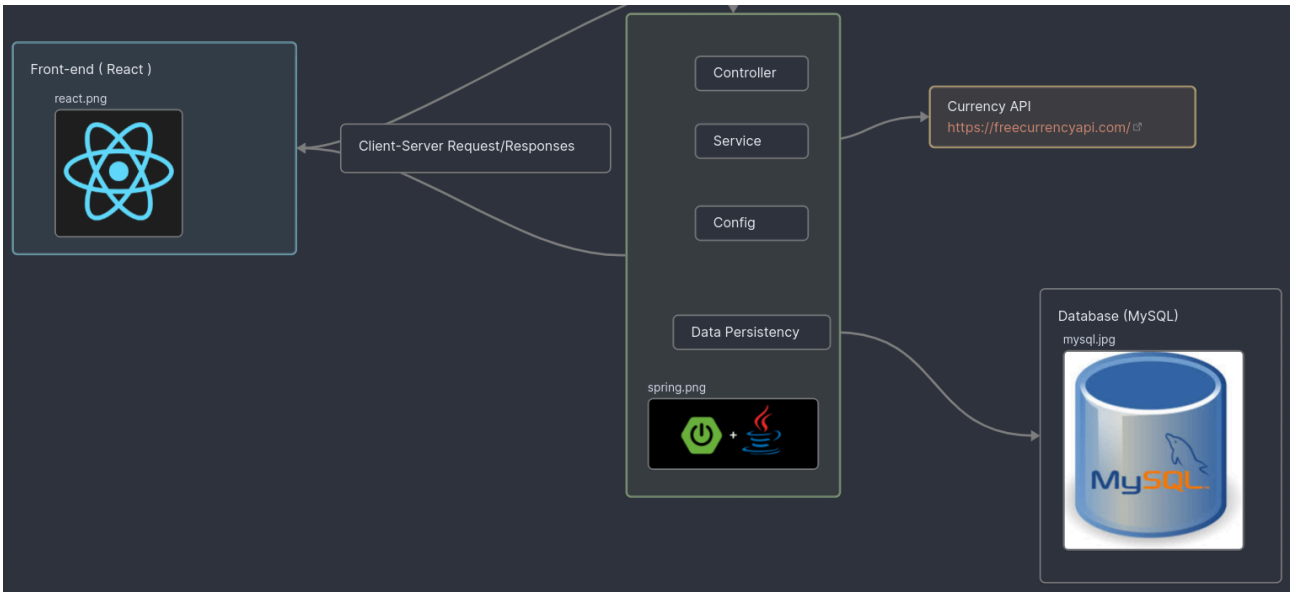
2.1 Functional scope and supported interactions

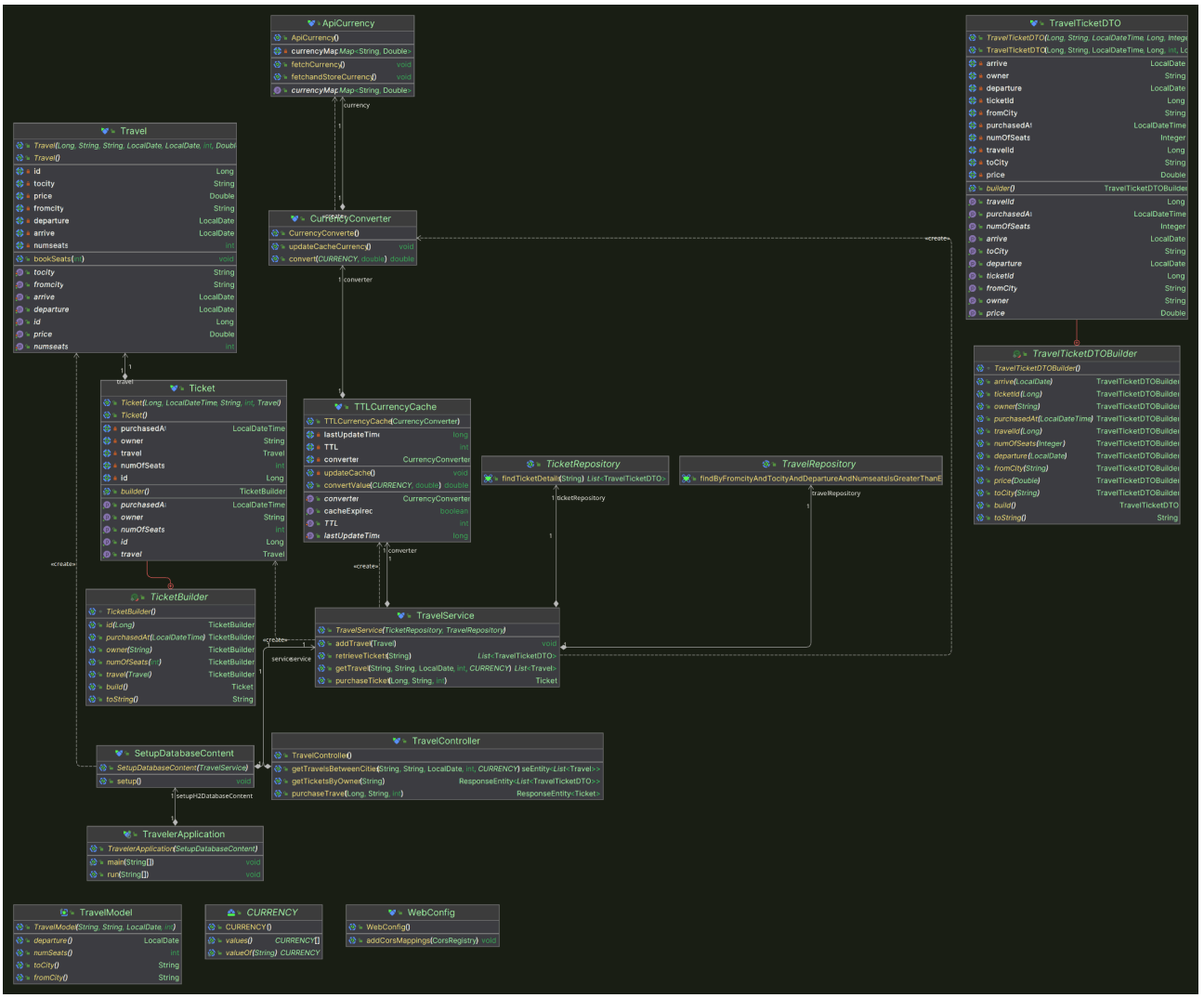
Currently, the application supports the search for a travel between two cities (by a given date) that has at least n seats available (like booking a ticket for n persons), purchase this travel and visualize the incoming travel tickets purchased. The user can choose which currency he can use to visualize the price, between Euro, Dollar, Iene and Pound

2.2 System architecture

The overall architecture is composed by :

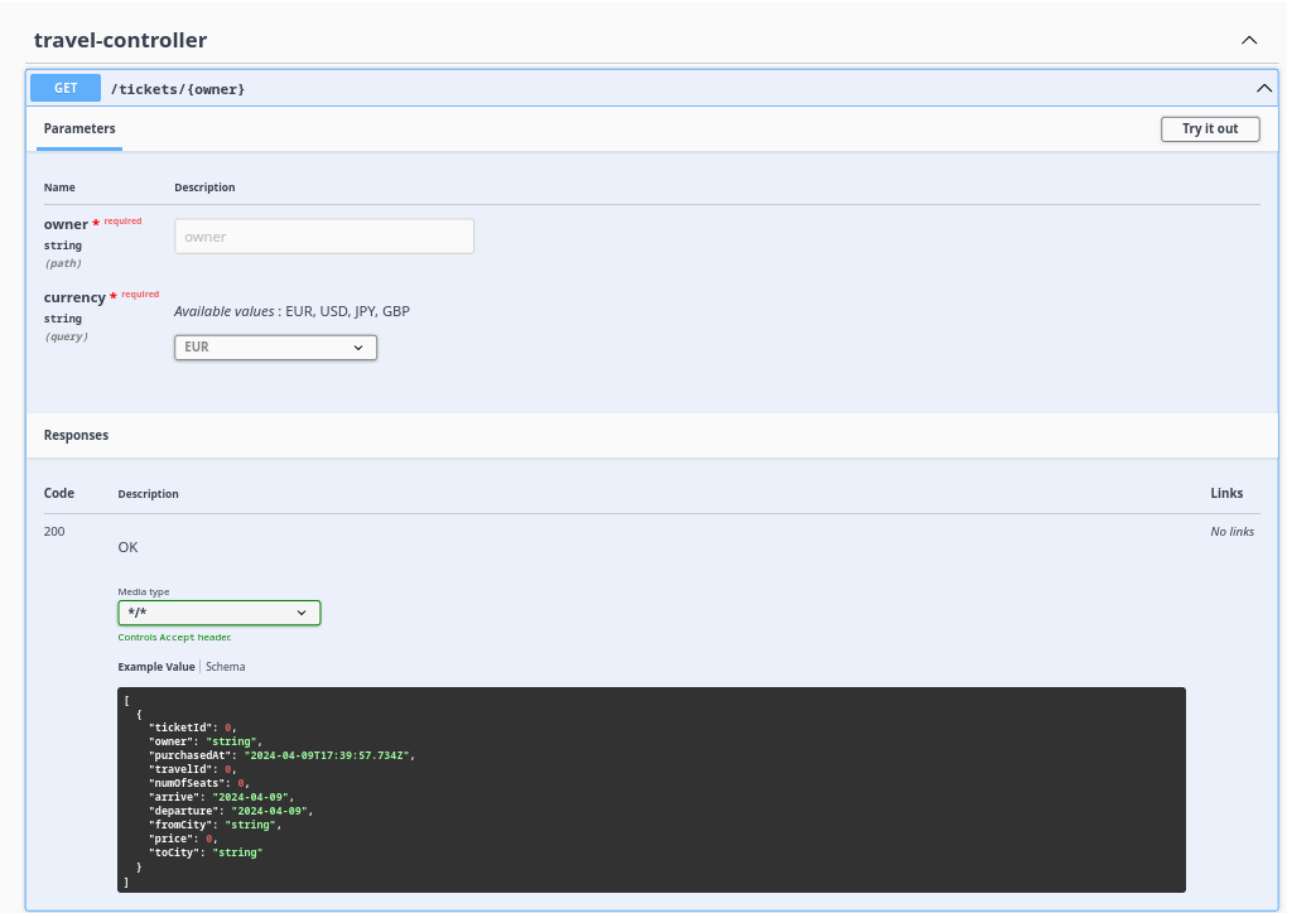
1. Front-end single-page web interface (React)
2. Back-end, holder of the service-side/logical/business of the application, that itself is composed tools like :
    a. Spring-boot frame work
    b. Lombok annotations
    c. Jpa support
    d. Testing tools : Junit 5, selenium, Mockito
    e. And others minors dependencies
3. MySQL Database, that support the two main tables, travel and tickets
4. External exchange rate on-line API (that is cached in the Back-end)



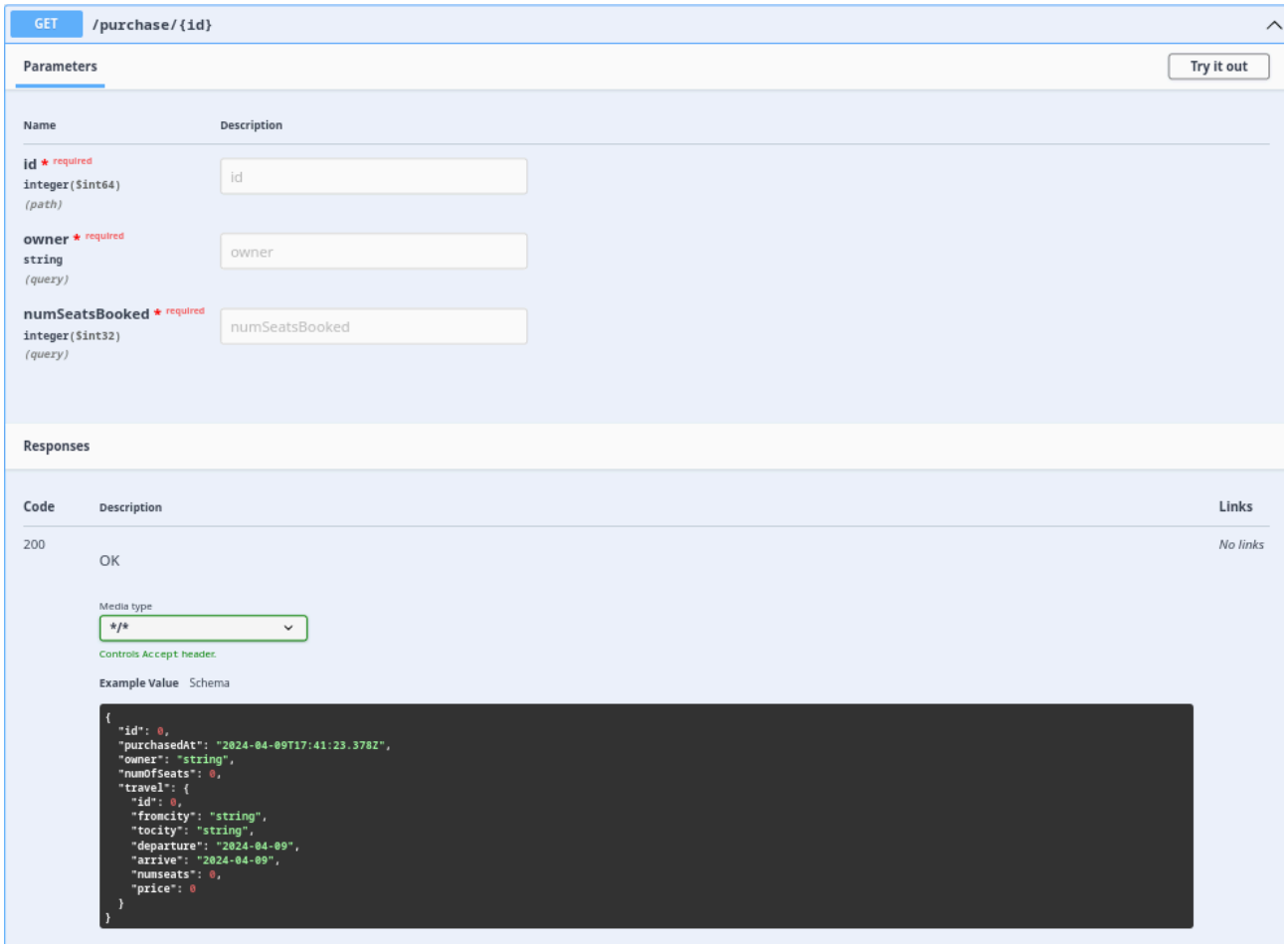

ApiCurrency
Travel
Ticket
CurrencyConverter
TTLCurrencyCache
TicketRepository
TravelRepository
TravelTicketDTO
TravelTicketDTOBuilder
TicketBuilder
TravelService
TravelController
SetupDatabaseContent
TravelerApplication
TravelModel
CURRENCY
WebConfig

2.3 API for developers

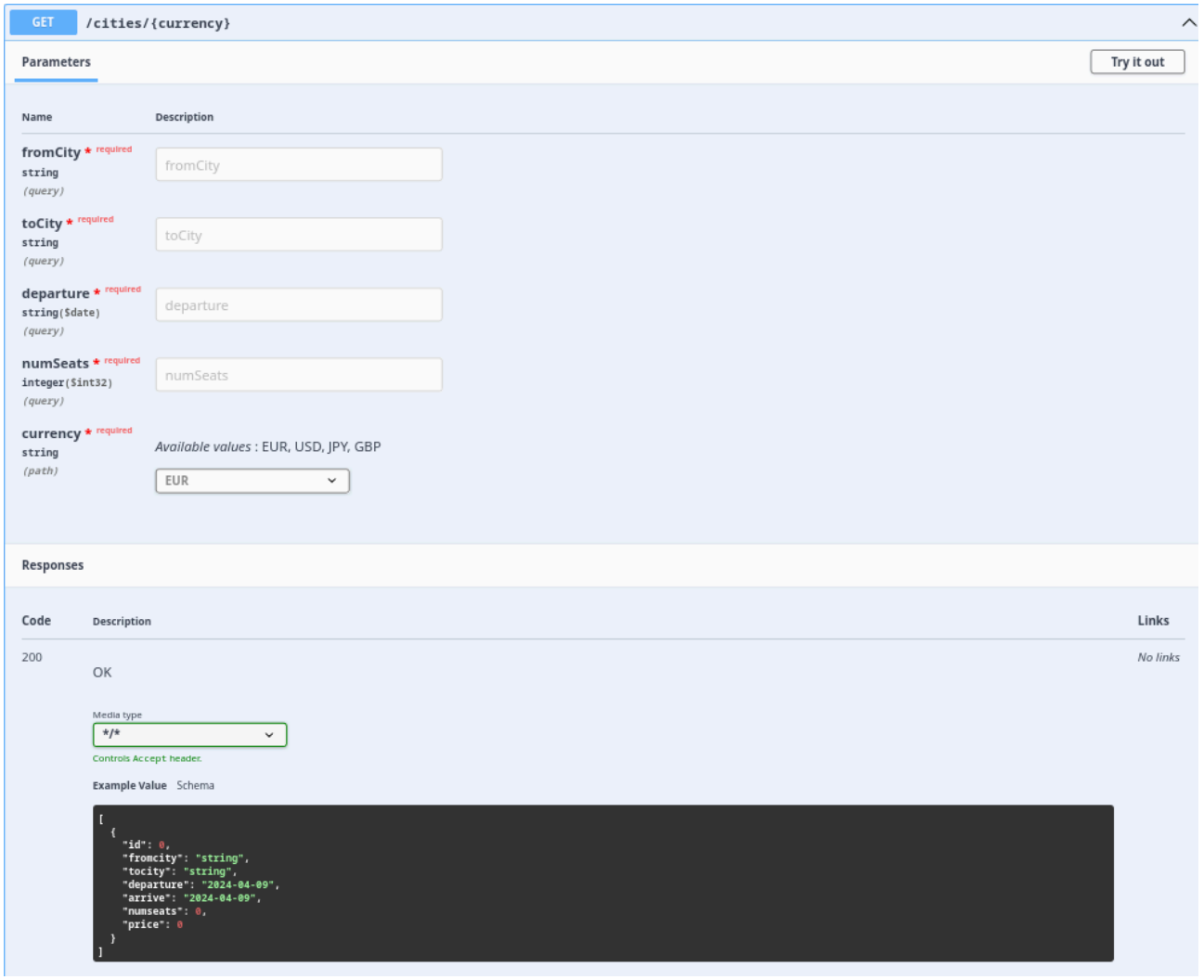
- The api has three endpoints :
  - One to retrieve all tickets bought by a given user

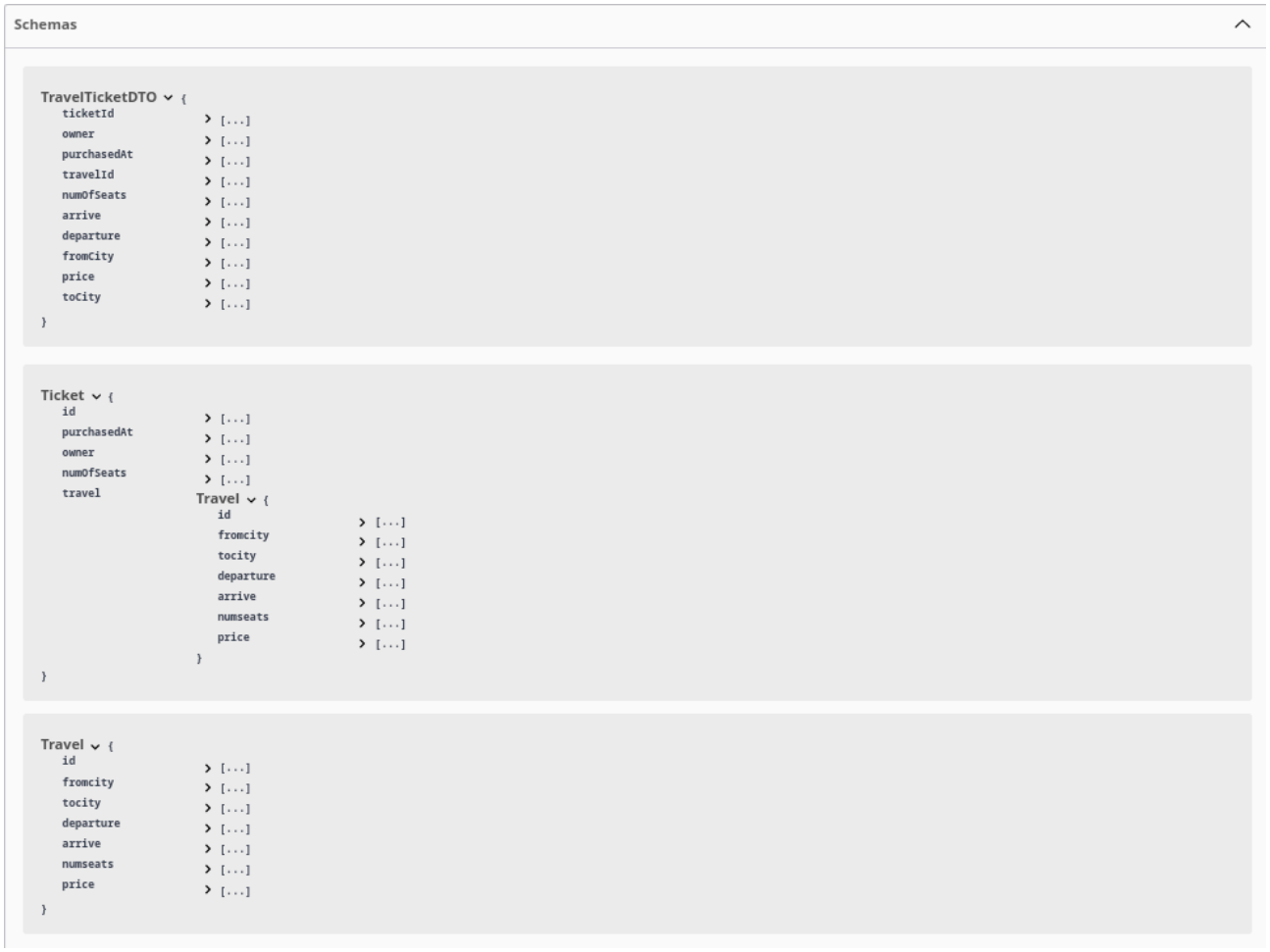


- One to purchase a ticket for a travel



- One to fetch all travels between two given cities, a given date and at least n seats available





3 Quality assurance

3.1 Overall strategy for testing

The tests were developed in order to verify the more common results of use of the api components (service layer, controller layer and external api communication layer), in order to ensure that each one of them were working normally and individually before going to a further layer.

As the api was being developed and each component were properly tested and verified, some additional tests were added to verify alternative scenarios, e.g. what a search component should return when an entity was not found in the fetch/mock data.

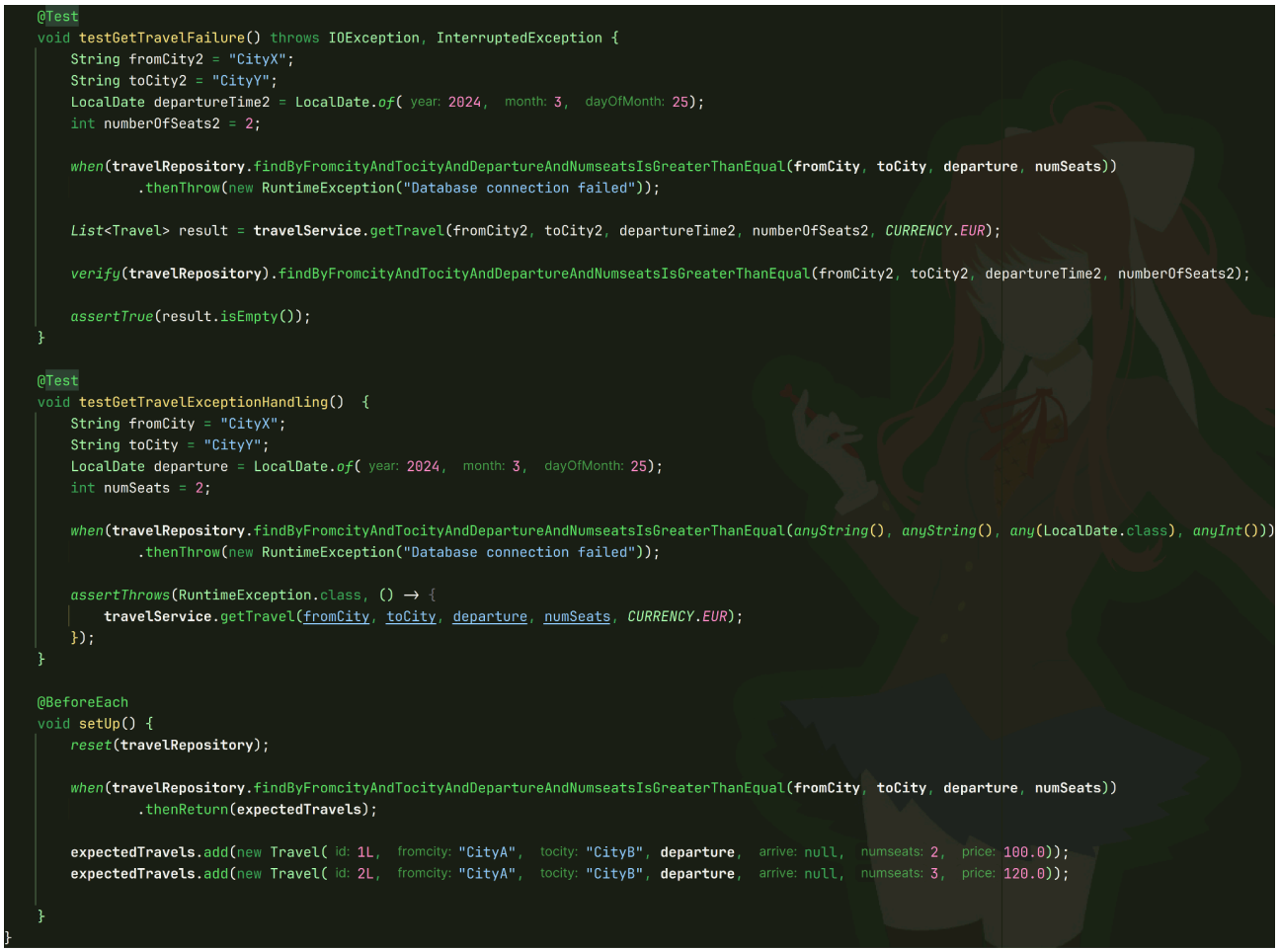
By the end of the development,a total of 16 tests (methods, not classes obviously) were written.

3.2 Unit and integration testing

Probably most of all backend were tested with tested, using unit test to verify the usual working of the service(logical layer) :

1 - Testing search mechanism of travels :

```java
@ExtendWith(MockitoExtension.class)
@MockitoSettings(strictness = Strictness.LENIENT)
class ServiceTest
{
    6 usages
    @Mock
    private TravelRepository travelRepository;

    3 usages
    @InjectMocks
    private TravelService travelService;
    4 usages
    String fromCity = "CityA";
    4 usages
    String toCity = "CityB";
    6 usages
    LocalDate departure = LocalDate.of( year: 2024,  month: 3,  dayOfMonth: 25);
    4 usages
    int numSeats = 2;


    4 usages
    List<Travel> expectedTravels = new ArrayList<>();


    @Test
    void testGetTravel() throws IOException, InterruptedException {
        List<Travel> result = travelService.getTravel(fromCity, toCity, departure, numSeats, CURRENCY.EUR);

        verify(travelRepository).findByFromcityAndTocityAndDepartureAndNumseatsIsGreaterThanEqual(fromCity, toCity, departure, numSeats);

        assertEquals(expectedTravels, result);
    }
```



```java
    @Test
    void testGetTravelFailure() throws IOException, InterruptedException {
        String fromCity2 = "CityX";
        String toCity2 = "CityY";
        LocalDate departureTime2 = LocalDate.of( year: 2024,  month: 3,  dayOfMonth: 25);
        int numberOfSeats2 = 2;

        when(travelRepository.findByFromcityAndTocityAndDepartureAndNumseatsIsGreaterThanEqual(fromCity, toCity, departure, numSeats))
                .thenThrow(new RuntimeException("Database connection failed"));

        List<Travel> result = travelService.getTravel(fromCity2, toCity2, departureTime2, numberOfSeats2, CURRENCY.EUR);

        verify(travelRepository).findByFromcityAndTocityAndDepartureAndNumseatsIsGreaterThanEqual(fromCity2, toCity2, departureTime2, numberOfSeats2);

        assertTrue(result.isEmpty());
    }

    @Test
    void testGetTravelExceptionHandling()  {
        String fromCity = "CityX";
        String toCity = "CityY";
        LocalDate departure = LocalDate.of( year: 2024,  month: 3,  dayOfMonth: 25);
        int numSeats = 2;

        when(travelRepository.findByFromcityAndTocityAndDepartureAndNumseatsIsGreaterThanEqual(anyString(), anyString(), any(LocalDate.class), anyInt()))
                .thenThrow(new RuntimeException("Database connection failed"));

        assertThrows(RuntimeException.class, () -> {
            travelService.getTravel(fromCity, toCity, departure, numSeats, CURRENCY.EUR);
        });
    }

    @BeforeEach
    void setUp() {
        reset(travelRepository);

        when(travelRepository.findByFromcityAndTocityAndDepartureAndNumseatsIsGreaterThanEqual(fromCity, toCity, departure, numSeats))
                .thenReturn(expectedTravels);

        expectedTravels.add(new Travel( id: 1L,  fromcity: "CityA",  tocity: "CityB", departure,  arrive: null,  numseats: 2,  price: 100.0));
        expectedTravels.add(new Travel( id: 2L,  fromcity: "CityA",  tocity: "CityB", departure,  arrive: null,  numseats: 3,  price: 120.0));

    }
}
```

2 - Testing purchasing mechanism of travels (tickets) :

```java
@ExtendWith(MockitoExtension.class)
class TicketServiceTest {

    3 usages
    @Mock
    private TicketRepository ticketRepository;

    2 usages
    @Mock
    private TravelRepository travelRepository;

    3 usages
    @InjectMocks
    private TravelService ticketService;


    2 usages
    @Captor
    private ArgumentCaptor<String> ownerCaptor;

    @Test
    void testPurchaseTicket() {
        // Mock data
        Long travelId = 1L;
        String owner = "John Doe";
        Travel travel = new Travel();
        travel.setId(travelId);
        travel.setNumseats(10);
        Optional<Travel> optionalTravel = Optional.of(travel);

        // Mock behavior
        when(travelRepository.findById(travelId)).thenReturn(optionalTravel);

        // Method call
        Ticket purchasedTicket = ticketService.purchaseTicket(travelId, owner,  numSeatsBooked: 1);

        // Verification
        assertNotNull(purchasedTicket);
        assertEquals(owner, purchasedTicket.getOwner());
        assertNotNull(purchasedTicket.getPurchasedAt());
        assertEquals(travel, purchasedTicket.getTravel()); // Assuming Ticket has appropriate equals() method

        verify(ticketRepository, times( wantedNumberOfInvocations: 1)).save(purchasedTicket);
        optionalTravel.get().bookSeats(
                numSeatsBooked: 1
        );
        verify(travelRepository, times( wantedNumberOfInvocations: 1)).save(optionalTravel.get());
    }
```

```java
@Test
void testRetrieveTickets() throws IOException, InterruptedException {
    String owner = "John Doe";
    List<TravelTicketDTO> expectedTickets = new ArrayList<>();

    TravelTicketDTO ticket1 = TravelTicketDTO.builder()
            .price(100.0).toCity("CityB").fromCity("CityA")
            .arrive( LocalDate.now()).departure( LocalDate.now()).ticketId(1L)
            .owner("James Lee")
            .travelId(1L).numOfSeats(2).purchasedAt(LocalDateTime.now())
            .build();

    expectedTickets.add(ticket1);

    when(ticketRepository.findTicketDetails(owner)).thenReturn(expectedTickets);

    List<TravelTicketDTO> actualTickets = ticketService.retrieveTickets(owner, CURRENCY.EUR);

    assertNotNull(actualTickets);
    assertEquals(expectedTickets.size(), actualTickets.size());
    for (int i = 0; i < expectedTickets.size(); i++) {
        assertEquals(expectedTickets.get(i).getTicketId(), actualTickets.get(i).getTicketId());
    }

    verify(ticketRepository).findTicketDetails(ownerCaptor.capture());
    assertEquals(owner, ownerCaptor.getValue());
}


@Test
void testPurchaseTicketWhenTravelNotFound() {
    Long nonExistentTravelId = 2L;
    String owner = "John Doe";

    Ticket purchasedTicket = ticketService.purchaseTicket(nonExistentTravelId, owner,  numSeatsBooked: 1);

    assertNull(purchasedTicket);
}
```

So was to test the external exchange-rate currency functionality, especially the Time-To-Live logic

```
class CurrencyCacheTest
{
    5 usages
    private TTLCurrencyCache cache;
    2 usages
    private CurrencyConverter converter;

    @Test
    //@Disabled
    void testCacheExpired() throws IOException, InterruptedException {
        Thread.sleep( millis: 6000);
        assertTrue(cache.isCacheExpired());
    }

    @Test
    //@Disabled
    void testCacheUpdate() throws IOException, InterruptedException {
        Thread.sleep( millis: 6000);
        cache.convertValue(CURRENCY.USD,  valueToBeConverted: 200);
        assertFalse(cache.isCacheExpired());
    }

    @Test
    void testFetchingData() {
        final Map<String, Double> currMap = converter.getCurrencyValues();

        for(CURRENCY currency : CURRENCY.values())
        {
            assertTrue(currMap.containsKey(currency.toString()));
        }
    }


    @BeforeEach
    public void setuptest() throws IOException, InterruptedException
    {
        this.cache = new TTLCurrencyCache(new CurrencyConverter());
        this.cache.setTTL(5000);
        this.converter = new CurrencyConverter();
    }

}
```

Persistency test for the database and repository, but this one is not fully developed yet:

```java
@ExtendWith(MockitoExtension.class)
@AutoConfigureMockMvc
@DataJpaTest
class TravelRepositoryTest {

    2 usages
    @Mock
    private TravelRepository travelRepository;

    @Test
     void testFindByFromCityAndToCityAndDepartureAndNumSeats() {
        String fromCity = "CityA";
        String toCity = "CityB";
        LocalDate departure = LocalDate.of( year: 2024,  month: 4,  dayOfMonth: 8);
        int numSeats = 5;

        Travel travel1 = new Travel( id: 1L,  fromcity: "CityA",  tocity: "CityB", departure, departure.plusDays( daysToAdd: 1),  numseats: 10,  price: 100.0);
        Travel travel2 = new Travel( id: 2L,  fromcity: "CityA",  tocity: "CityB", departure, departure.plusDays( daysToAdd: 1),  numseats: 8,  price: 90.0);
        List<Travel> expectedTravels = Arrays.asList(travel1, travel2);

        when(travelRepository.findByFromcityAndTocityAndDepartureAndNumseatsIsGreaterThanEqual(fromCity, toCity, departure, numSeats))
                .thenReturn(expectedTravels);

        List<Travel> foundTravels = travelRepository.findByFromcityAndTocityAndDepartureAndNumseatsIsGreaterThanEqual(fromCity, toCity, departure, numSeats);

        assertEquals( expected: 2, foundTravels.size());
        assertEquals( expected: 1L, foundTravels.get(0).getId());
        assertEquals( expected: 2L, foundTravels.get(1).getId());
    }
}
```

Lastly, but not less important, the integration test focusing in ensuring the central working the controller :

```java
@AutoConfigureMockMvc
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.MOCK)
@RunWith(SpringRunner.class)
class ControllerTest
{
    @Autowired
    private MockMvc mockController;

    @MockBean
    private TravelService service;

    6 usages
    private final Travel DummyTravel = new Travel( id: 1L,  fromcity: "Dublin, Ireland", tocity: "Galway, Ireland",LocalDate.now(),  LocalDate.now(),  numseats: 6,  price: 11.99);

    2 usages
    private final Ticket DummyTicket = Ticket.builder()
            .owner("James Lee")
            .travel(DummyTravel)
            .purchasedAt(LocalDateTime.now())
            .build();


    @Test
    void testSearchByGivenCities() throws Exception {...}

    @Test
    void testGetTravelsBetweenCitiesNotFound() throws Exception {...}



    @Test
    void testPurchaseTicketForATravel() throws Exception {

        // 'http://localhost:9090/purchase/1?owner=JohnDoe&numSeatsBooked=2'

        MvcResult result = mockController.perform(get( urlTemplate: "/purchase/1")
                        .param( name: "owner",  ...values: "James Lee")
                        .param( name: "numSeatsBooked",  ...values: "2")
                        .contentType(MediaType.APPLICATION_JSON))
                .andDo(print())
                .andExpect(status().isOk())
                .andReturn();

        int statusCode = result.getResponse().getStatus();
        assertEquals( expected: 200, statusCode);
    }
```

```java
@Test
void testGetTicketsByOwner() throws Exception {...}


@BeforeEach
void setupTest() throws IOException, InterruptedException {
    final List<Travel> trips = new ArrayList<>();
    trips.add(DummyTravel);

    when(service.purchaseTicket(
            eq(DummyTravel.getId()),
            eq(DummyTicket.getOwner()),
            anyInt()
    )).thenReturn(DummyTicket);

    when(service.getTravel(
            eq(DummyTravel.getFromcity()),
            eq(DummyTravel.getTocity()),
            any(LocalDate.class),
            eq(DummyTravel.getNumseats()),
            eq(CURRENCY.EUR)
    )).thenReturn(trips);
    TravelTicketDTO td1 = TravelTicketDTO.builder()
            .price(11.99).toCity("Galway, Ireland").fromCity("Dublin, Ireland")
            .arrive( LocalDate.now()).departure( LocalDate.now()).ticketId(1L)
            .owner("JohnDoe")
            .travelId(1L).numOfSeats(2).purchasedAt(LocalDateTime.now())
            .build();

    TravelTicketDTO td2 = TravelTicketDTO.builder()
            .price(11.99).toCity("London, UK").fromCity("Paris, France")
            .arrive( LocalDate.now()).departure( LocalDate.now()).ticketId(2L)
            .owner("JohnDoe")
            .travelId(2L).numOfSeats(1).purchasedAt(LocalDateTime.now())
            .build();

    List<TravelTicketDTO> tickets = Arrays.asList(
            td1,
            td2
    );
    when(service.retrieveTickets(anyString(), any(CURRENCY.class))).thenReturn(tickets);


    String fromCity = "Paris, France";
    String toCity = "Madrid, Spain";
    int numSeats = 6;

    when(service.getTravel(eq(fromCity), eq(toCity), any(LocalDate.class), eq(numSeats), any(CURRENCY.class)))
            .thenReturn(Collections.emptyList());
}
```

3.3 Functional testing

There were written functional tests with Selenium and Cucumber, unfortunately, due to time issues, the last one is not fully implemented and is left to be future work.

Selenium tests were written to test two features specifically :

1. Assert the selection of a specific currency (by default its used EUR) in the front-end, which will be used to the conversion operation in the backend

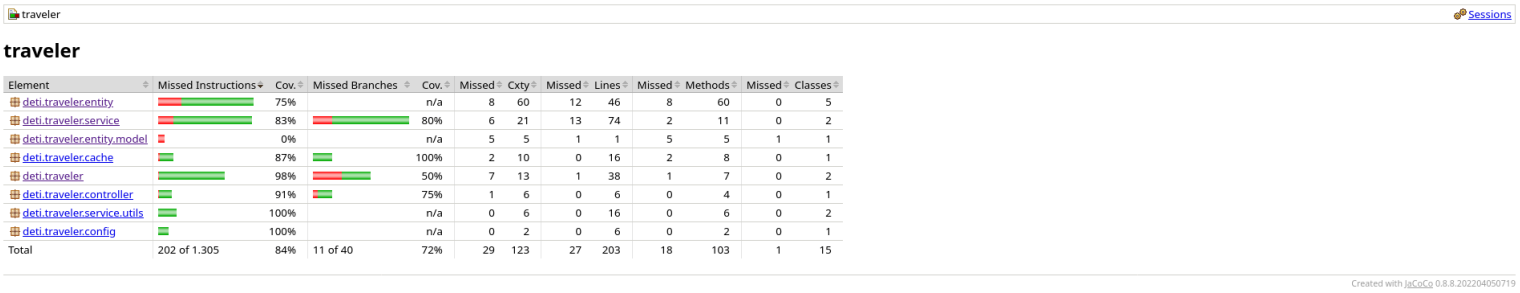
```java
@ExtendWith(SeleniumJupiter.class)
@SpringBootTest
@Disabled
class CurrencySelectionTest {

    17 usages
    private WebDriver driver;

    1 usage
    private final String currency = "GBP";

    @BeforeEach
    public void startDriver(FirefoxDriver ddriver) {
        driver = ddriver;
        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait( time: 5, TimeUnit.SECONDS);
    }

    @AfterEach
    public void killDriver() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    void testSelenium() {
            // Open the URL
            driver.get("http://localhost:5173");

            driver.manage().window().setSize(new Dimension( width: 1898,  height: 1004));

            driver.findElement(By.cssSelector("label:nth-child(1) > input")).sendKeys( ...keysToSend: "Aveiro");
            driver.findElement(By.cssSelector("label:nth-child(2) > input")).sendKeys( ...keysToSend: "Paris");
            driver.findElement(By.cssSelector("label:nth-child(4) > input")).click();
            driver.findElement(By.cssSelector("label:nth-child(4) > input")).sendKeys( ...keysToSend: "2024-07-11");
            driver.findElement(By.cssSelector("label:nth-child(5) > input")).sendKeys( ...keysToSend: "2");

            driver.findElement(By.cssSelector(".travel-list")).click();
            driver.findElement(By.cssSelector(".css-tj5bde-Svg")).click();
            driver.findElement(By.id("react-select-3-option-3")).click();
            driver.findElement(By.cssSelector("button")).click();

            WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(5));
            WebElement currencyShow = wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("selectedCurrencySho

            System.out.println("/n Currency choose : " + currencyShow.getText());

            // Asserting text using JUnit assertions
            assertEquals(currency, currencyShow.getText(),  message: "Currency text does not match");


    }
}
```

2. And test ensure the presence of the purchase of a ticket (the last booked) in a list in the react front end

```java
@ExtendWith(SeleniumJupiter.class)
@SpringBootTest
@Slf4j
@Disabled
class PurchaseTest {

    22 usages
    private WebDriver driver;

    2 usages
    private final String fromCity = "Aveiro, Aveiro", toCity = "Albergaria das Cabras, Arouca", date = "2024-05-20", numOfSeats = "3";


    @BeforeEach
    public void startDriver(FirefoxDriver ddriver) {
        driver = ddriver;
        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait( time: 5, TimeUnit.SECONDS);
    }

    @AfterEach
    public void killDriver() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    public void testSelenium() {
        try {
            driver.get("http://localhost:5173");

            driver.manage().window().setSize(new Dimension( width: 1898,  height: 1004));

            driver.findElement(By.cssSelector("label:nth-child(1) > input")).sendKeys(fromCity);
            driver.findElement(By.cssSelector("label:nth-child(2) > input")).sendKeys(toCity);
            driver.findElement(By.cssSelector("label:nth-child(4) > input")).click();
            driver.findElement(By.cssSelector("label:nth-child(4) > input")).sendKeys(date);

            driver.findElement(By.cssSelector(".css-w9q2zk-Input2")).click();
            driver.findElement(By.id("react-select-3-option-2")).click();
            driver.findElement(By.cssSelector("button")).click();

            WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(15));
            WebElement purchaseButton = wait.until(ExpectedConditions.visibilityOfElementLocated(By.cssSelector(".travel-item:nth-child(1) .buttonItem")));

            purchaseButton.click();

            WebElement numSeatsField = driver.findElement(By.id("numSeats"));
            numSeatsField.clear();
            numSeatsField.sendKeys( ...keysToSend: "2");

            // Double click to clear the field
            Actions actions = new Actions(driver);
            actions.doubleClick(numSeatsField).perform();

            // Fill the number of seats field again
            numSeatsField.sendKeys(numOfSeats);

            // Click on a button
            driver.findElement(By.cssSelector(".buttonItem:nth-child(11)")).click();

            WebElement fromCityLabel = driver.findElement(By.cssSelector("tr:last-child > td:nth-child(2)"));
            WebElement toCityLabel = driver.findElement(By.cssSelector("tr:last-child > td:nth-child(3)"));
            WebElement numSeatsSelected = driver.findElement(By.cssSelector("tr:last-child > td:nth-child(6)"));

            assertEquals(fromCity,fromCityLabel.getText(),  message: "The last purchase's from city should be Aveiro, Aveiro" );
            assertEquals(toCity,toCityLabel.getText(),  message: "The last purchase's to city should be Albergaria das Cabras, Arouca" );
            assertEquals(numOfSeats,numSeatsSelected.getText(),  message: "The last purchase's number of seats purchased should be 3" );

        } finally {
            driver.quit();
        }
    }
}
```

3.4 Code quality analysis

Firstly, the regular report of jacoco specify that the overall project of the backend has a total of 84% of code coverage, which implies that there isn't much of non-tested code, the 16% remaining mostly are code parts that were left because they are, or at least could be, useful in (hypothetical)



traveler

Sessions

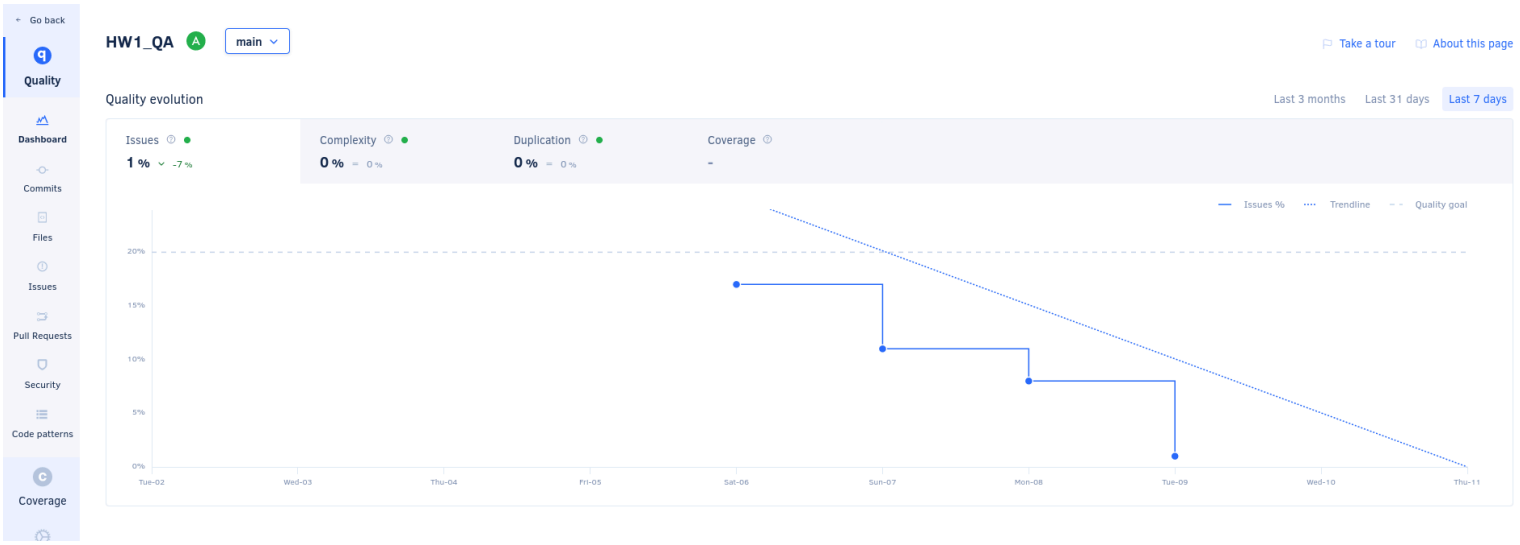
traveler

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
deti.traveler.entity		75%		n/a	8	60	12	46	8	60	0	5
deti.traveler.service		83%		80%	6	21	13	74	2	11	0	2
deti.traveler.entity.model		0%		n/a	5	5	1	1	5	5	1	1
deti.traveler.cache		87%		100%	2	10	0	16	2	8	0	1
deti.traveler		98%		50%	7	13	1	38	1	7	0	2
deti.traveler.controller		91%		75%	1	6	0	6	0	4	0	1
deti.traveler.service.utils		100%		n/a	0	6	0	16	0	6	0	2
deti.traveler.config		100%		n/a	0	2	0	6	0	2	0	1
Total	202 of 1.305	84%	11 of 40	72%	29	123	27	203	18	103	1	15

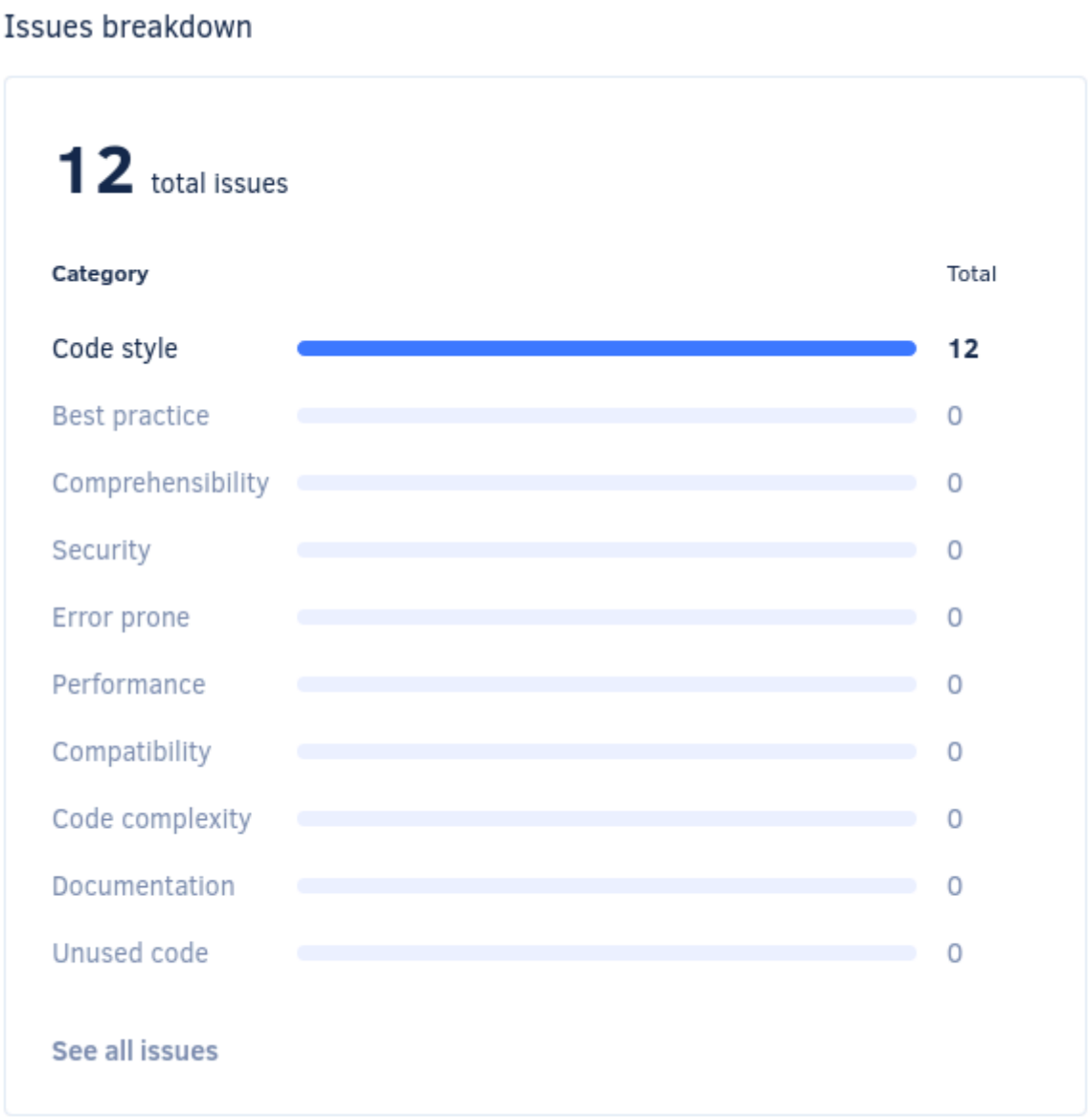
Created with JaCoCo 0.8.8.202204050719

future features, especially the code parts in the service.

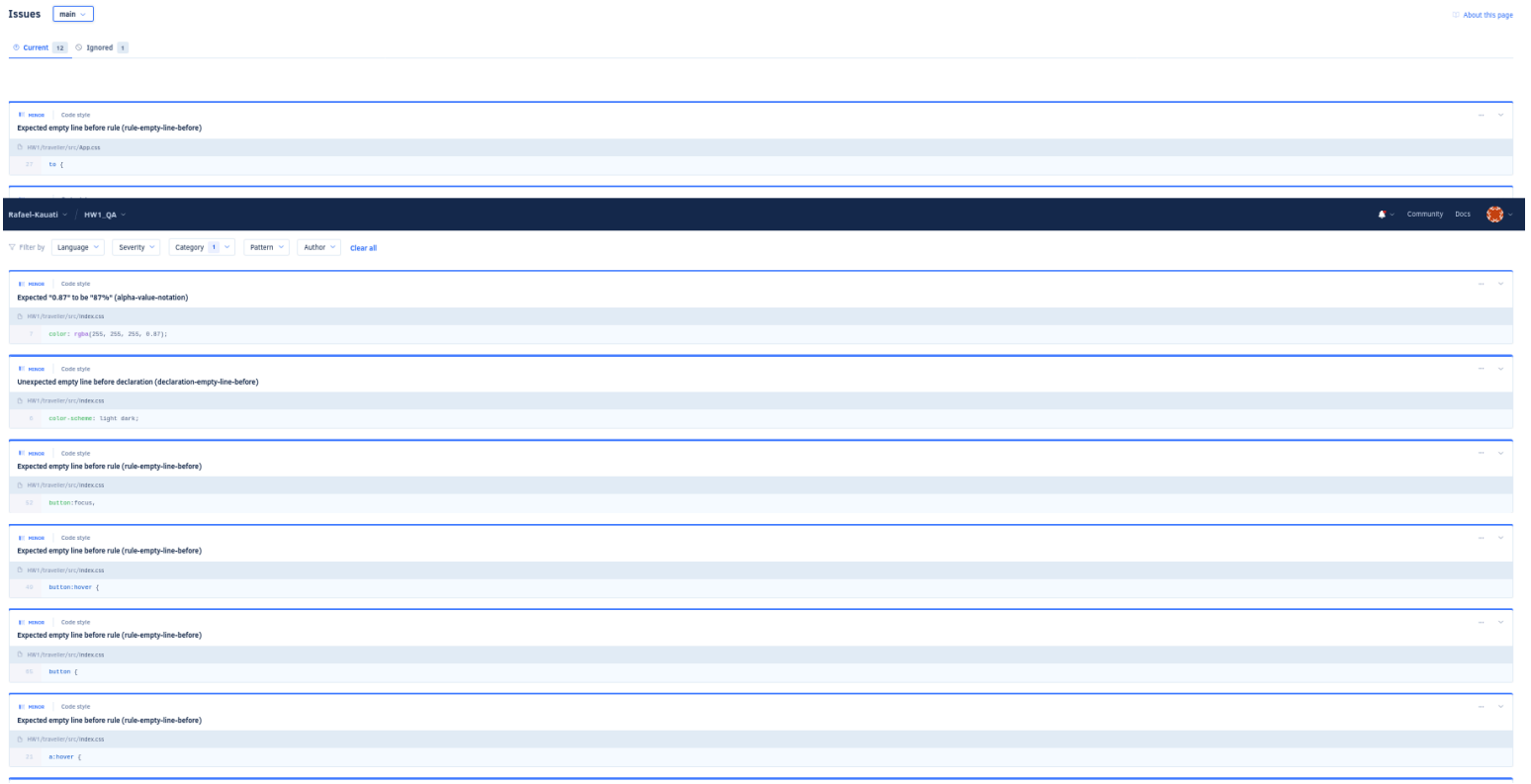
Now using a more professional QA tool, codacy analysis was used to track code style issues, error prones, security and other quality metrics. By the moment where 70% of the features and code were done and tested, it was started to track and correct the found issues :



Leaving only a total of 12 code style issues :



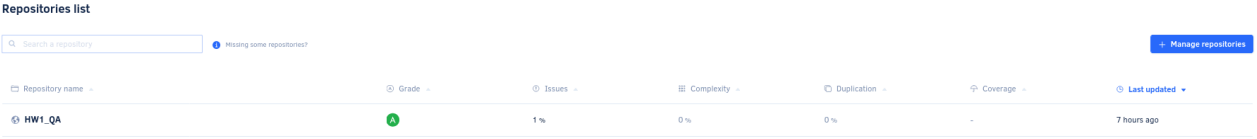
Since these code style issues only relate to .css files(that dont really affect logical code reading), they could be “ignored” in the issue correction, over the project development

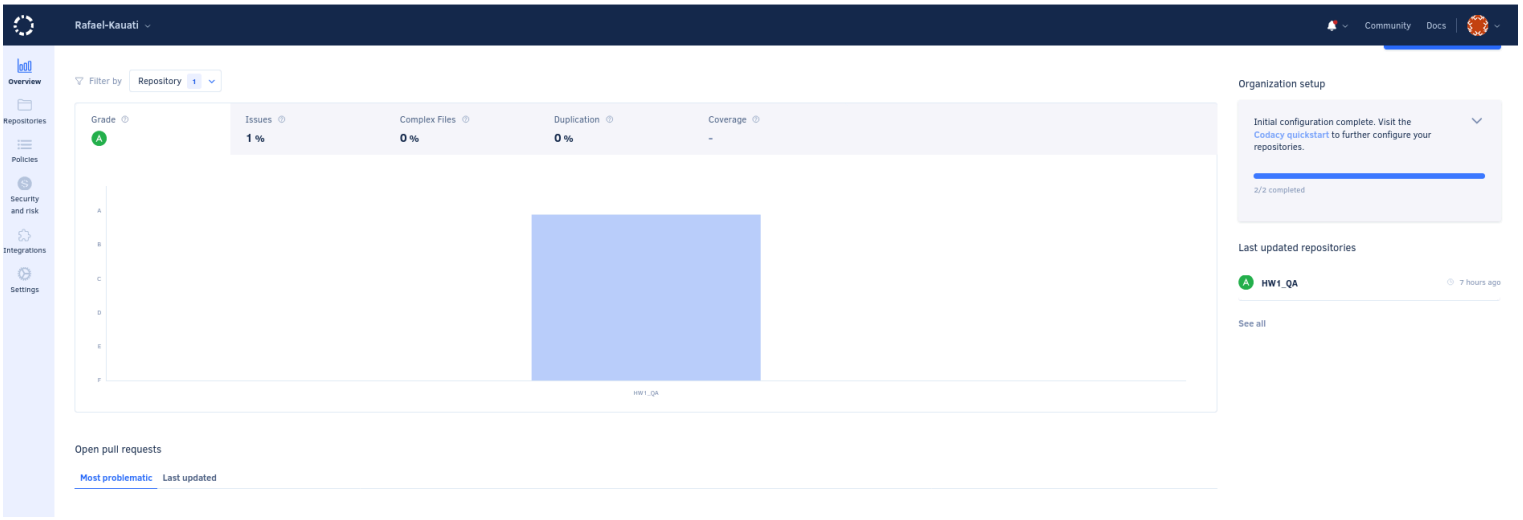


During these issues correction, most of them were simply naming pattern issues,unused dependency imports and unused constructors , and one of them was a subject of security regarding how was handled the external currency api keys of access, in the ApiCurrency.java class, that particularly took some time to resolve, another one that took more time to understand why it was marked as an issue was making test classes visibility package level.

Most of the correction, especially when it came to coverage analysis, was to add/remove lombok annotations that could have or not unused additional fields in classes.

From the beginning of the development , the codacy quality grading was “B”, mostly due to the number of style codes and some error prone issues, and also one security/vulnerability issue . At the end of the project development, it was possible to raise the overall grading to “A”





Quality management of code is something that should be dealt with from the beginning of the project, since you never know when it's going to need a specific component/module of code by a peer, dry code is always welcome due to readability issues.

3.5 Continuous integration pipeline [optional]

Unfortunately, due to time issues, an continuos integration pipeline was not possible to be implemented properly until the deadline, leaving to be developed in future work

4 References & resources

Project resources

Resource:	URL/location:
Git repository	https://github.com/Rafael-Kauati/TQS_105925/tree/main/HW1
Video demo	https://streamable.com/dca72y

	(Also present in HW1/ directory in my portfolio github repository)
QA dashboard (online)	[**optional**; if you have a quality dashboard available online (e.g.: sonarcloud), place the URL here]
CI/CD pipeline	[**optional**; if you have th CI pipeline definition in a server, place the URL here]
Deployment ready to use	[**optional**; if you have the solution deployed and running in a server, place the URL here]

Reference materials

- https://start.spring.io/
- https://softwareengineering.stackexchange.com/questions/220053/why-did-java-make-package-access-default
- https://www.baeldung.com/spring-boot-testing
- https://www.codacy.com/
- https://en.wikipedia.org/wiki/Code_coverage
- https://freecurrencyapi.com/