COSC2440 – Assignment 2 – Build a backend

CAR BOOKING SYSTEM

Technical report

Supervised by: Mr Vu Thanh Minh

Nguyen Luu Quoc Bao *(s3877698)*

Dao Kha Tuan *(s3877347)*

Bui Quang An *(s3877482)*

Nguyen Trong Minh Long *(s3878694)*

Table of Contents

INTRODUCTION

On-demand car book applications have no longer been a strange term to phone users [1]. Those applications are user-friendly and accessible [1]. However, the backend sides of them are operated with complicated architecture. This technical report illustrates how a typical car booking system works in tandem with deeply providing a technical view of the application architecture.

BUSSINESS REQUIREMENT

The stakeholder requires a comprehensive backend for administrators and users to use through Application Programming Interfaces (APIs). The requirement can be separated into six modules, including: manage car, manage driver, manage customer, manage invoice, manage booking, and manage revenue. Those operations authorize the administrators to manually adjust the system. Another noticeable point is that each entity must have the created date time for database usage.

Mange car

Create-Read-Update-Delete (CRUD) operations are required on the car. Car information comprises vehicle identification number (VIN), make, model, color, convertible, rating, license plate, and rate per kilometer. Cars can also be filtered by VIN, license plate, make, and model to make the app more user-friendly. Each car will be taken by a driver. Furthermore, the admin can view the days that cars are used in a month by inputting a month in a year. The server will respond with a table of results in this case.

Manage driver

Create-Read-Update-Delete (CRUD) operations are required on the car. Each driver has an id, a license number, a phone number, and a rating. A driver can be searched by id, phone number, or license number. The driver also has an API to choose his/her car. This car can be changed at any time. However, a driver can only choose one or no car at once.

Manage customer

Create-Read-Update-Delete (CRUD) operations are required on the customer. Each customer has an id, name, phone number, and address. Customers can be filtered by id, name, phone number, and address to maximize the user experience. Customers has book car functions, which will be discussed on the User book car section.

Manage booking

Create-Read-Update-Delete (CRUD) operations are required on the customer. When a booking is managed, the invoice will be affected, except for the read operation. Booking can be filtered by id, period, or date. When a user book a car, a booking will be automatically created.

Manage invoice

Create-Read-Update-Delete (CRUD) operations are required on the invoice. When an invoice is managed, the booking stays unchanged when the invoice is managed. Invoices can also be filtered by period and date, which is similar to booking operation.

Manage revenue

Additionally, stakeholders want to sum up revenue for financial purposes. Therefore, the app should have functions filter all revenue (sum of all total charges in invoice), revenue by period, revenue by a customer on a period, revenue by a driver on a period.

Customer book car

The customer can book a car. When they book, the system will display a list of available cars. The users can pick any car that they want.

USECASE DIAGRAM

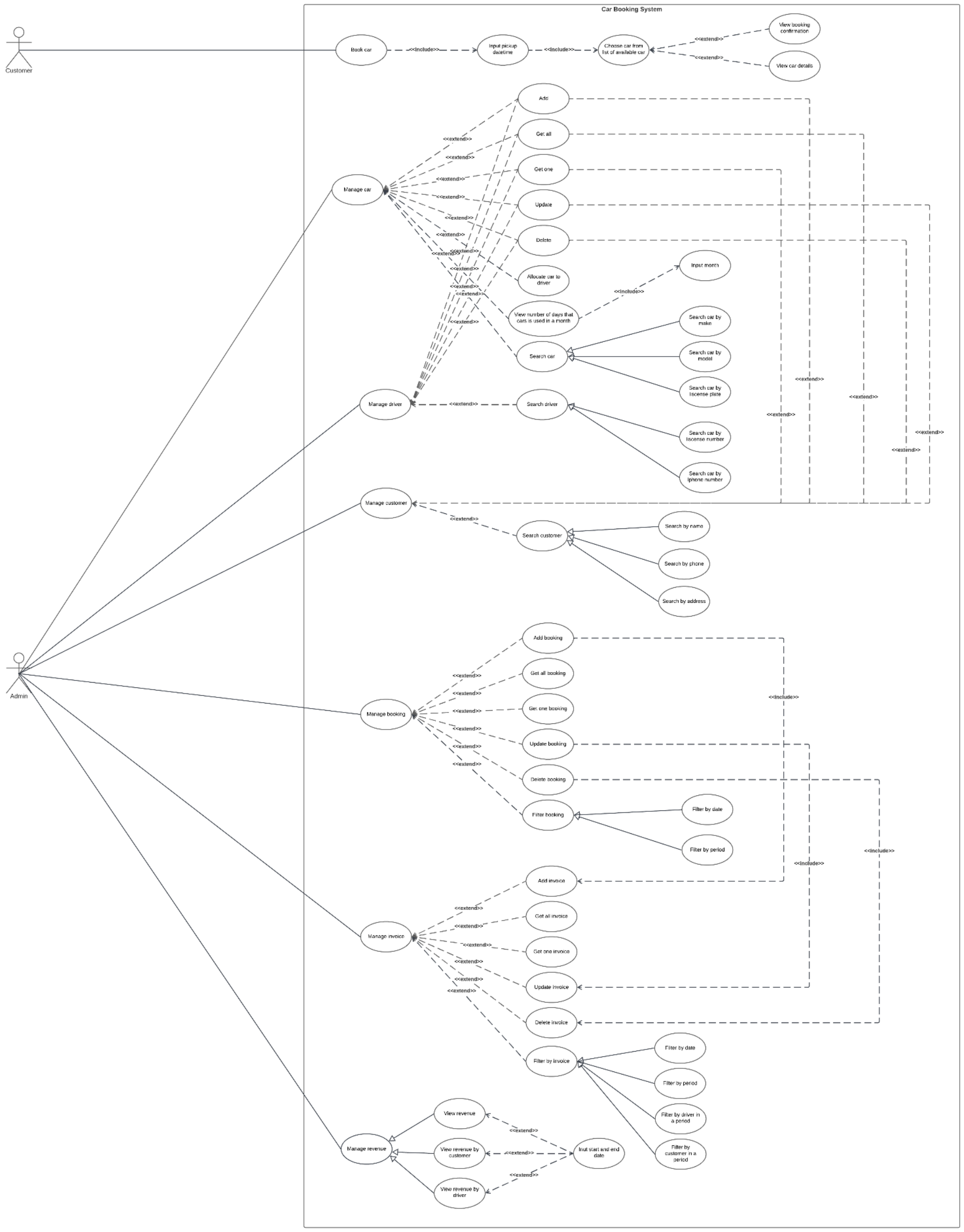


Figure 1. Usecase diagram (PDF is attached on uml directory)

TECHNOLOGY USED

Version Control System

Our group has decided to use GitHub as a tool to help keep track of every member's work as well as to have a share place to save the share work. In addition to that, using GitHub, our group can resolve conflict as well as to check earlier version of code so that in case the code fail, our group still has the earlier version to check or to use as preference.

Language Use

As one of the requirements of the project, which is a Java backend implementation for taxi company, our group choose to use Java version 16.0.1 to write the codes and implement the program.

Spring Boot

Our group chooses to use Spring Boot because Spring Boots make it easier to create a stand-alone, production-grade Spring based Applications that we can just simply run the program. In addition, Spring Boot can automatically configure Spring and 3rd party libraries whenever possible and needs no code generation and no requirement for XML configuration.

PostgreSQL

As a free and open-source relation database management, we choose PostgreSQL to host our primary database and use it with Spring JPA as an access layer to the main database.

CRUD Repository

Our group chooses to use the CRUD repository instead of the Session Factory so that the code is much cleaner and more convenient. Moreover, using the CRUD repository helps prevent unnecessary bugs and potential conflict among sessions.

Spring Hibernate

Our group finds it easier to use Spring Hibernate because Hibernate use Java Database Connectivity. In other words, Hibernate allows the configuration of PostgreSQL access easier.

Spring Web MVC

Because our group project requirements are to develop a backend for the taxi company, Spring Web MVC is the one that satisfies the requirements of our project. To make it clearer, Spring MVC follows the Model-View-Controller design pattern. This way the web browser can call the function so that it can return the value and display it on the web browser.

IDE

As for the software or editor for our code, we use IntelliJ ultimate edition to create, generate, and config spring project as well as write our code.

Testing

As for testing our methods, functions, and models of our program, we have two different methods of testing.

- First, we use JUnit Testing 4 to test all our functions and methods to see if we have missed any cases or our code return the right value or if the behaviour of the code is expected just like our group has planned.
- Second, to test our APIs, we use Postman API to test each of our function to see if it behaves just as we intended to.

Meeting

As for meeting and discussion for our project, our group uses Microsoft Teams to set meeting and have discussion about the project. We choose to use Teams because it is more formal and it can store earlier meeting transcript and video so that if any team member has missed it, he or she can watch it again later.

Diagrams

For diagrams such as use-case diagram and class diagram, our group used Lucid Chart. Not only does it allow the user to draw complex diagram, but it also allows the user to share and work on the same files so that the diagram can be consistency. Moreover, Lucid Chart allow user to export the diagram into PDF file, which is very convenient.

ARCHITECTURE

Model-View-Controller

In 1985, Model-View-Controller (MVC) design pattern concept was first introduced and used in the Smalltalk-80 programming environment [2]. After many years, it has been developed in many different ways and has become one of the most popular design patterns applied in most web applications [2].

MVC consists of three components including "Model", "View" and "Controller" and each of them plays an important role in the web. When the web browser sends the HTTP request to the server, the Controller will take and handle the request [3]. Based on the request, the Controller will send it to the Model component and the model will interact with the database and return the results to the Controller [3]. After that, the results will continue to be sent to the "View" component from the Controller to be rendered before sending back to the Controller and the Controller will respond to the interface containing the data to the web browser [3].

In our project, we have used the MVC design pattern to build the taxi management system. So far, we have made the "Model" component by creating entities to interact with the table in the PostgreSQL database, the "Controller" component by creating controller classes to interact with each entity and handle the request from web browser, specifically Postman. We have not built the "View" component to render the data due to the lack of time and the requirements of the project.

Repository pattern

In order to make the Business logic layer, which is the service layer, contact with the data source, one of the most important things we must do is create the abstraction layer from the repository pattern.
The Repository pattern is an intermediate layer that enables services to interact with the database by passing available functions through abstraction layers from the repository pattern [4]. In our project, we use PageingAndSortingRepository to create repository abstraction layers for each entity.

Service layer pattern

One of the most vital parts of MVC architecture we cannot forget to mention is the service layer pattern. The service layer pattern helps developers apply their logic to make communication between the controller and the repository [5]. Moreover, by getting and handling the data, service layer pattern also plays the role as a bridge connecting the data source and the presentation layers, which is the user interface (UI). In our project, there are 5 entities and each of them has its own service so that the repository's methods can be used, and the controller can interact with the entity.

CLASS DIAGRAM

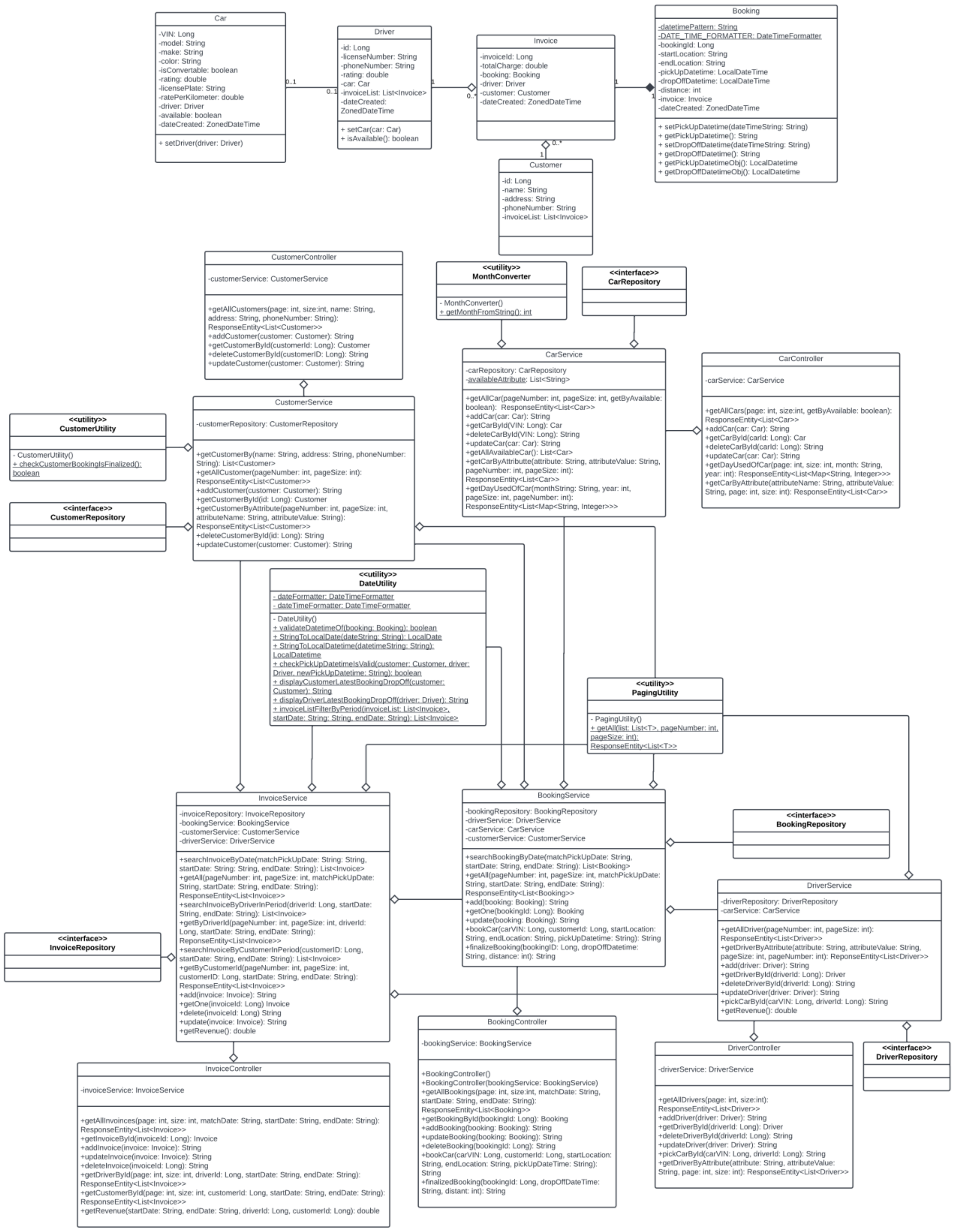

Car
-VIN: Long
-model: String
-make: String
-color: String
-isConvertable: boolean
-rating: double
-licensePlate: String
-ratePerKilometer: double
-driver: Driver
-available: boolean
-dateCreated: ZonedDateTime
+ setDriver(driver: Driver)
0..1
0..1
Driver
-id: Long
-licenseNumber: String
-phoneNumber: String
-rating: double
-car: Car
-invoiceList: List<Invoice>
-dateCreated: ZonedDateTime
+ setCar(car: Car)
+ isAvailable(): boolean
1
0..*
Invoice
-invoiceId: Long
-totalCharge: double
-booking: Booking
-driver: Driver
-customer: Customer
-dateCreated: ZonedDateTime
1
1
Booking
-datetimePattern: String
-DATE_TIME_FORMATTER: DateTimeFormatter
-bookingId: Long
-startLocation: String
-endLocation: String
-pickUpDatetime: LocalDateTime
-dropOffDatetime: LocalDateTime
-distance: int
-invoice: Invoice
-dateCreated: ZonedDateTime
+ setPickUpDatetime(dateTimeString: String)
+ getPickUpDatetime(): String
+ setDropOffDatetime(dateTimeString: String)
+ getDropOffDatetime(): String
+ getPickUpDatetimeObj(): LocalDatetime
+ getDropOffDatetimeObj(): LocalDatetime
0..*
1
Customer
-id: Long
-name: String
-address: String
-phoneNumber: String
-invoiceList: List<Invoice>
CustomerController
-customerService: CustomerService
+getAllCustomers(page: int, size:int, name: String, address: String, phoneNumber: String): ResponseEntity<List<Customer>>
+addCustomer(customer: Customer): String
+getCustomerById(customerId: Long): Customer
+deleteCustomerById(customerID: Long): String
+updateCustomer(customer: Customer): String
<<utility>>
MonthConverter
- MonthConverter()
+ getMonthFromString(): int
<<interface>>
CarRepository
CarService
-carRepository: CarRepository
-availableAttribute: List<String>
+getAllCar(pageNumber: int, pageSize: int, getByAvailable: boolean): ResponseEntity<List<Car>>
+addCar(car: Car): String
+getCarById(VIN: Long): Car
+deleteCarById(VIN: Long): String
+updateCar(car: Car): String
+getAllAvailableCar(): List<Car>
+getCarByAttribute(attribute: String, attributeValue: String, pageNumber: int, pageSize: int): ResponseEntity<List<Car>>
+getDayUsedOfCar(monthString: String, year: int, pageSize: int, pageNumber: int): ResponseEntity<List<Map<String, Integer>>>
CarController
-carService: CarService
+getAllCars(page: int, size:int, getByAvailable: boolean): ResponseEntity<List<Car>>
+addCar(car: Car): String
+getCarById(carId: Long): Car
+deleteCarById(carId: Long): String
+updateCar(car: Car): String
+getDayUsedOfCar(page: int, size: int, month: String, year: int): ResponseEntity<List<Map<String, Integer>>>
+getCarByAttribute(attributeName: String, attributeValue: String, page: int, size: int): ResponseEntity<List<Car>>
<<utility>>
CustomerUtility
- CustomerUtility()
+ checkCustomerBookingIsFinalized(): boolean
<<interface>>
CustomerRepository
CustomerService
-customerRepository: CustomerRepository
+getCustomerBy(name: String, address: String, phoneNumber: String): List<Customer>
+getAllCustomer(pageNumber: int, pageSize: int): ResponseEntity<List<Customer>>
+addCustomer(customer: Customer): String
+getCustomerById(id: Long): Customer
+getCustomerByAttribute(pageNumber: int, pageSize: int, attributeName: String, attributeValue: String): ResponseEntity<List<Customer>>
+deleteCustomerById(id: Long): String
+updateCustomer(customer: Customer): String
<<utility>>
DateUtility
- dateFormatter: DateTimeFormatter
- dateTimeFormatter: DateTimeFormatter
- DateUtility()
+ validateDatetimeOf(booking: Booking): boolean
+ StringToLocalDate(dateString: String): LocalDate
+ StringToLocalDatetime(datetimeString: String): LocalDatetime
+ checkPickUpDatetimeIsValid(customer: Customer, driver: Driver, newPickUpDatetime: String): boolean
+ displayCustomerLatestBookingDropOff(customer: Customer): String
+ displayDriverLatestBookingDropOff(driver: Driver): String
+ invoiceListFilterByPeriod(invoiceList: List<Invoice>, startDate: String, endDate: String): List<Invoice>
<<utility>>
PagingUtility
- PagingUtility()
+ getAll(list: List<T>, pageNumber: int, pageSize: int): ResponseEntity<List<T>>
InvoiceService
-invoiceRepository: InvoiceRepository
-bookingService: BookingService
-customerService: CustomerService
-driverService: DriverService
+searchInvoiceByDate(matchPickUpDate: String, startDate: String, endDate: String): List<Invoice>
+getAll(pageNumber: int, pageSize: int, matchPickUpDate: String, startDate: String, endDate: String): ResponseEntity<List<Invoice>>
+searchInvoiceByDriverInPeriod(driverId: Long, startDate: String, endDate: String): List<Invoice>
+getByDriverId(pageNumber: int, pageSize: int, driverId: Long, startDate: String, endDate: String): ReponseEntity<List<Invoice>>
+searchInvoiceByCustomerInPeriod(customerID: Long, startDate: String, endDate: String): List<Invoice>
+getByCustomerId(pageNumber: int, pageSize: int, customerID: Long, startDate: String, endDate: String): ResponseEntity<List<Invoice>>
+add(invoice: Invoice): String
+getOne(invoiceId: Long) Invoice
+delete(invoiceId: Long) String
+update(invoice: Invoice): String
+getRevenue(): double
BookingService
-bookingRepository: BookingRepository
-driverService: DriverService
-carService: CarService
-customerService: CustomerService
+searchBookingByDate(matchPickUpDate: String, startDate: String, endDate: String): List<Booking>
+getAll(pageNumber: int, pageSize: int, matchPickUpDate: String, startDate: String, endDate: String): ResponseEntity<List<Booking>>
+add(booking: Booking): String
+getOne(bookingId: Long): Booking
+update(booking: Booking): String
+bookCar(carVIN: Long, customerId: Long, startLocation: String, endLocation: String, pickUpDatetime: String): String
+finalizeBooking(bookingID: Long, dropOffDatetime: String, distance: int): String
<<interface>>
BookingRepository
DriverService
-driverRepository: DriverRepository
-carService: CarService
+getAllDriver(pageNumber: int, pageSize: int): ResponseEntity<List<Driver>>
+getDriverByAttribute(attribute: String, attributeValue: String, pageSize: int, pageNumber: int): ReponseEntity<List<Driver>>
+add(driver: Driver): String
+getDriverById(driverId: Long): Driver
+deleteDriverById(driverId: Long): String
+updateDriver(driver: Driver): String
+pickCarById(carVIN: Long, driverId: Long): String
+getRevenue(): double
<<interface>>
InvoiceRepository
BookingController
-bookingService: BookingService
+BookingController()
+BookingController(bookingService: BookingService)
+getAllBookings(page: int, size:int, matchDate: String, startDate: String, endDate: String): ResponseEntity<List<Booking>>
+getBookingById(bookingId: Long): Booking
+addBooking(booking: Booking): String
+updateBooking(booking: Booking): String
+deleteBooking(bookingId: Long): String
+bookCar(carVIN: Long, customerId: Long, startLocation: String, endLocation: String, pickUpDateTime: String): String
+finalizedBooking(bookingId: Long, dropOffDateTime: String, distant: int): String
DriverController
-driverService: DriverService
+getAllDrivers(page: int, size:int): ResponseEntity<List<Driver>>
+addDriver(driver: Driver): String
+getDriverById(driverId: Long): Driver
+deleteDriverById(driverId: Long): String
+updateDriver(driver: Driver): String
+pickCarById(carVIN: Long, driverId: Long): String
+getDriverByAttribute(attribute: String, attributeValue: String, page: int, size: int): ResponseEntity<List<Driver>>
<<interface>>
DriverRepository
InvoiceController
-invoiceService: InvoiceService
+getAllInvoinces(page: int, size: int, matchDate: String, startDate: String, endDate: String): ResponseEntity<List<Invoice>>
+getInvoiceById(invoiceId: Long): Invoice
+addInvoice(invoice: Invoice): String
+updateInvoice(invoice: Invoice): String
+deleteInvoice(invoiceId: Long): String
+getDriverById(page: int, size: int, driverId: Long, startDate: String, endDate: String): ResponseEntity<List<Invoice>>
+getCustomerById(page: int, size: int, customerId: Long, startDate: String, endDate: String): ResponseEntity<List<Invoice>>
+getRevenue(startDate: String, endDate: String, driverId: Long, customerId: Long): double

Figure 2. Class diagram (PDF is attached on uml directory)

IMPLEMENTATION RESULT

We established multiple APIs to conduct Create – Read – Update – Delete (CRUD) operations on all the system's required entities: Car, Booking, Customer, Driver, and Invoice, to meet the business requirements. By applying PagingAndSortingRepository to our repositories, we implement pagination on all data returned from APIs so that it can be presented to users in pages for Read operations on each entity. Furthermore, we implement search APIs on each entity to search the data based on specific attributes, as well as statistical APIs to access each entity's statistical data.

The APIs for admin will have an endpoint called "admin", while the APIs for customer do not include it. The description of admin and customer's APIs can be briefly explained as below (the detailed description of the APIs is included in README.md file):

Role	Entity	Method	Usage
Admin	Booking	GET	Get all bookings filtered by a specific date or by a period. The data returned will be paginated.
		POST	Create a new booking or finalize an existing booking.
		PUT	Update booking
		DELETE	Delete an existing booking
	Invoice	GET	Get all invoices filtered by a specific date or by a period. The admin can also get the revenue of all invoices by a period, by a customer or driver. The data returned will be paginated.
		POST	Create a new invoice
		PUT	Update invoice
		DELETE	Delete an existing invoice
	Car	GET	Get the number of days a car was used within a month. The data returned will be paginated.
		POST	Create a new car
		PUT	Update a car
		DELETE	Delete an existing car
	Driver	GET	Get all drivers, get driver by ID or by attributes. The data returned will be paginated.
		POST	Create a new driver
		PUT	Update a driver
		DELETE	Delete an existing driver
	Customer	GET	Get all customer, get customer by ID or by attributes. The data returned will be paginated.
		POST	Create a new customer
		PUT	Update a customer
		DELETE	Delete an existing customer

Customer user	Booking	GET	Get a booking by ID
		POST	Book a car
	Car	GET	Get all cars, get car by ID, by availability or by attributes. The data returned will be paginated.
Driver	Driver	GET	Pick a car for a driver

To implement the pagination for all entities, we implemented our services to return an object of ResponseEntity. This object will take the input parameters including page number and size from HTTP requests and perform pagination on the data returned from the database. The data returned will be paginated based on the page number and size that users have input in the HTTP request parameters.

For the implementation of search APIs, we retrieve all the data of an entity from the database and filter them by the attributes and conditions that users have specified in the HTTP request parameters. The data returned will also be paginated by returning a ResponseEntity object with the page number and size included in HTTP request parameters.

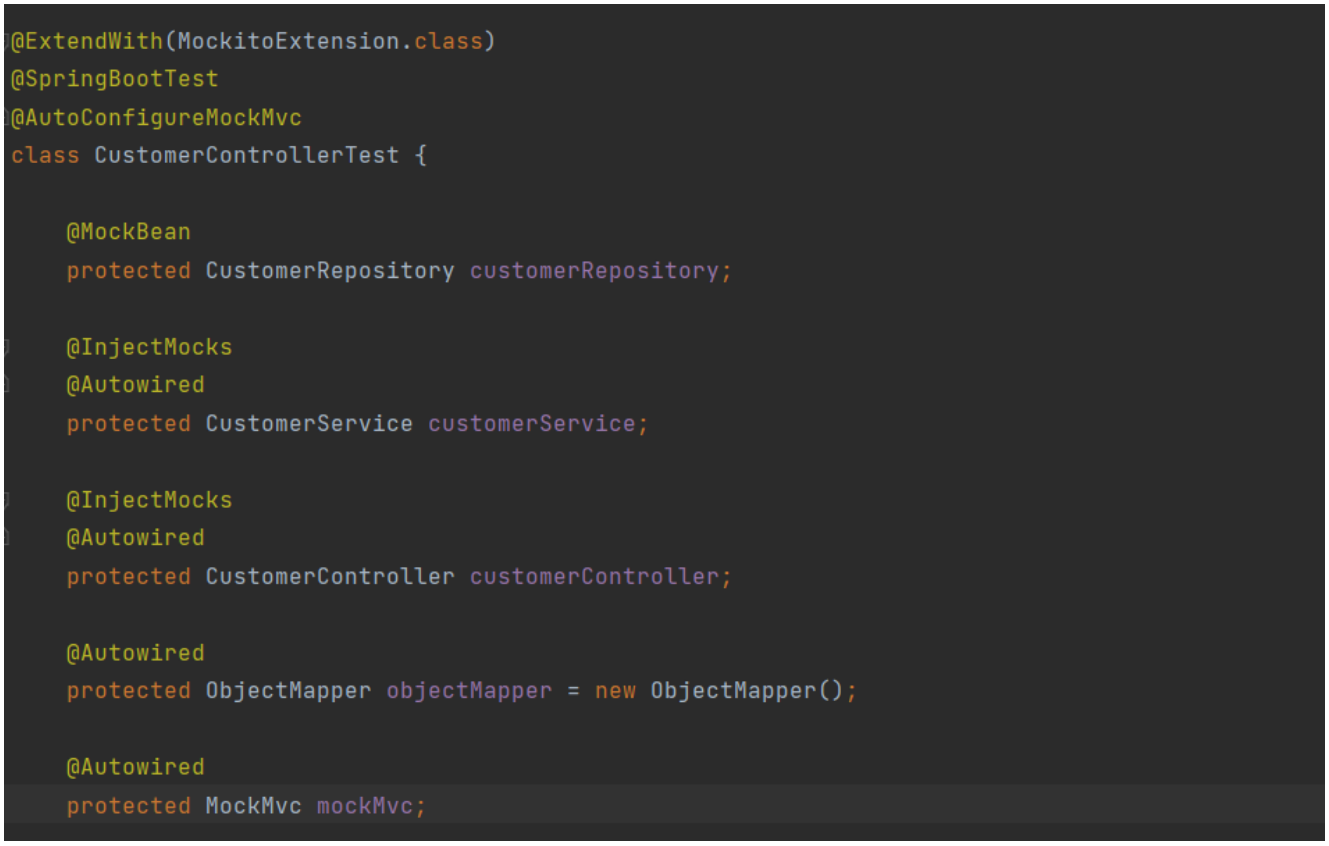
In the implementation of statistical APIs for invoices, we obtain a list of invoices by a period, by a certain customer or driver, then compute the total of all their total charges to produce a revenue that is required by the admin. However, to use the statistical API for cars, we first establish a map with the key being a car's license plate and the value being the number of days in a month that the car was driven. We iterate through all bookings created in a certain month, as provided by the admin in the HTTP request parameter, to determine how many days a car was driven during that month. The returned data is also paginated with ResponseEntity object.

TESTING

Process

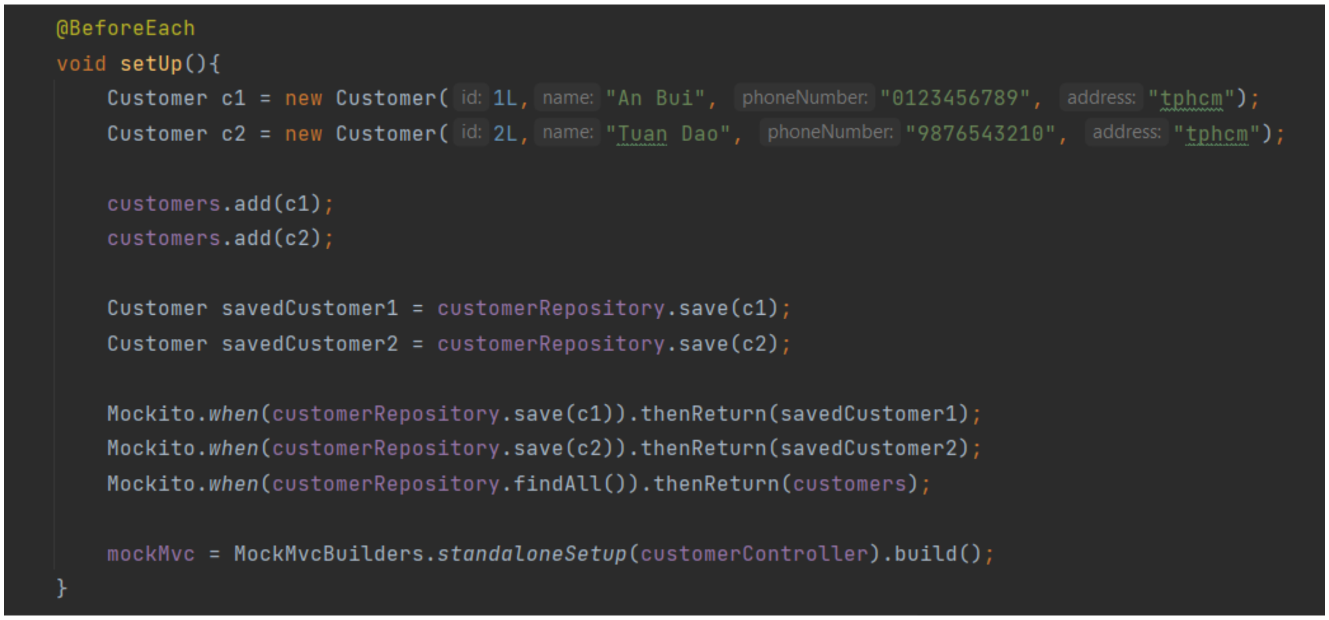
After developing the RESTful API, our group began to write testing scripts to make sure each method in each controller runs well. In order to test methods, using dependencies including JUnit4, Mockito and MockMVC is indispensable and very important. In each testing method, we divided it into two stages which are unit testing and integration testing.

To begin with, unit testing is the test of service layer, which means we test methods in services to check that our logic when building the functionality included in the API will not conflict with the database when the API is running. In each controller, the first thing we need to do is wiring the related Service and Repository by using the annotations called @MockBean and @Autowired. After that, we will construct the controller's MockMVC by using the standaloneSetup() function and set up the data used to test the methods



```
@ExtendWith(MockitoExtension.class)
@SpringBootTest
@AutoConfigureMockMvc
class CustomerControllerTest {

    @MockBean
    protected CustomerRepository customerRepository;

    @InjectMocks
    @Autowired
    protected CustomerService customerService;

    @InjectMocks
    @Autowired
    protected CustomerController customerController;

    @Autowired
    protected ObjectMapper objectMapper = new ObjectMapper();

    @Autowired
    protected MockMvc mockMvc;
```

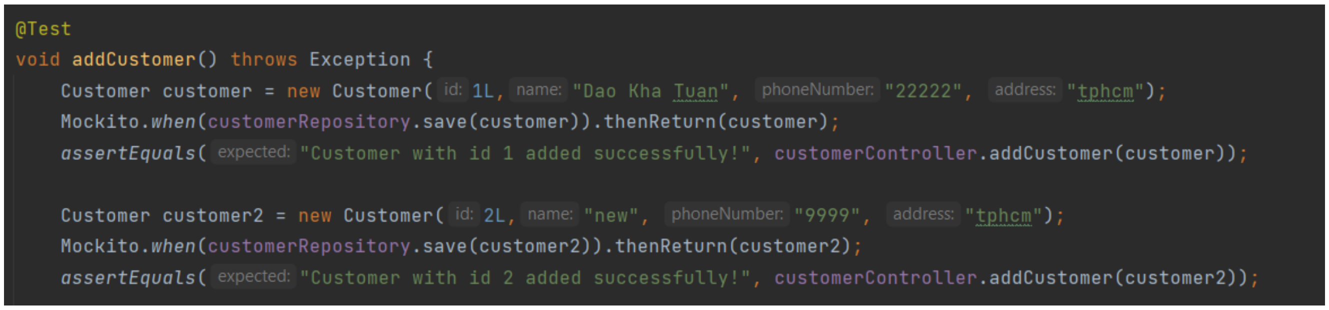
Figure 3. CustomerControllerTest class (Set up classes related to the test)



```
    @BeforeEach
    void setUp(){
        Customer c1 = new Customer( id: 1L, name: "An Bui", phoneNumber: "0123456789", address: "tphcm");
        Customer c2 = new Customer( id: 2L, name: "Tuan Dao", phoneNumber: "9876543210", address: "tphcm");

        customers.add(c1);
        customers.add(c2);

        Customer savedCustomer1 = customerRepository.save(c1);
        Customer savedCustomer2 = customerRepository.save(c2);

        Mockito.when(customerRepository.save(c1)).thenReturn(savedCustomer1);
        Mockito.when(customerRepository.save(c2)).thenReturn(savedCustomer2);
        Mockito.when(customerRepository.findAll()).thenReturn(customers);

        mockMvc = MockMvcBuilders.standaloneSetup(customerController).build();
    }
```

Figure 4. Set up the data and controller's MockMVC

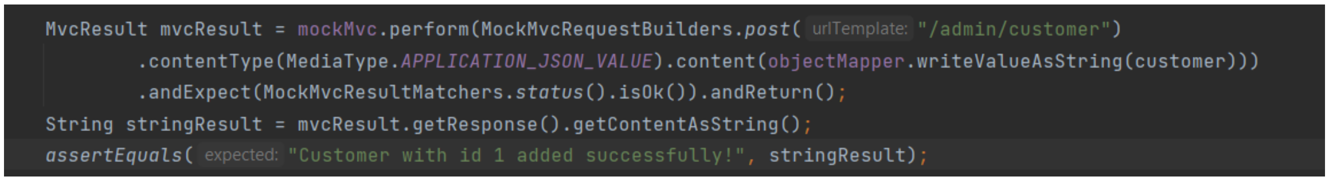
Next, we created the test cases for each method by mocking the repository, used methods in controller class and checked if output was same as we assumed.



```
@Test
void addCustomer() throws Exception {
    Customer customer = new Customer( id: 1L, name: "Dao Kha Tuan", phoneNumber: "22222", address: "tphcm");
    Mockito.when(customerRepository.save(customer)).thenReturn(customer);
    assertEquals( expected: "Customer with id 1 added successfully!", customerController.addCustomer(customer));

    Customer customer2 = new Customer( id: 2L, name: "new", phoneNumber: "9999", address: "tphcm");
    Mockito.when(customerRepository.save(customer2)).thenReturn(customer2);
    assertEquals( expected: "Customer with id 2 added successfully!", customerController.addCustomer(customer2));
```

Figure 5. addCustomer method (Unit test)

Finally, we did the integration testing, which is about testing the response when we send the HTTP request to the server by using the MockMVC.

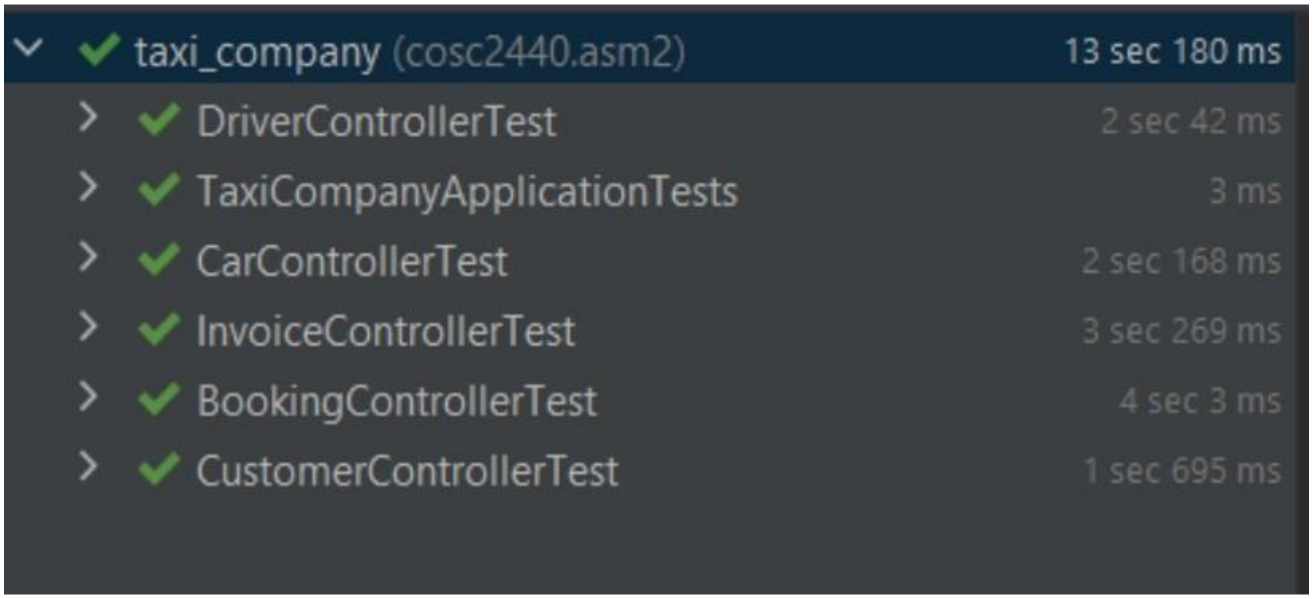


```
MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.post( urlTemplate: "/admin/customer")
        .contentType(MediaType.APPLICATION_JSON_VALUE).content(objectMapper.writeValueAsString(customer)))
        .andExpect(MockMvcResultMatchers.status().isOk()).andReturn();
String stringResult = mvcResult.getResponse().getContentAsString();
assertEquals( expected: "Customer with id 1 added successfully!", stringResult);
```

Figure 6. addCustomer method (Integration test)

Results

After writing unit tests for each controller, we have run all unit tests and it has worked successfully with a total of about 39 successful test cases of 5 controllers including different positive and negative cases in each method, we confirm that all test cases have run successfully, without any errors and with 100% coverage.



```
✓ taxi_company (cosc2440.asm2)        13 sec 180 ms
  ✓ DriverControllerTest               2 sec 42 ms
  ✓ TaxiCompanyApplicationTests        3 ms
  ✓ CarControllerTest                  2 sec 168 ms
  ✓ InvoiceControllerTest              3 sec 269 ms
  ✓ BookingControllerTest              4 sec 3 ms
  ✓ CustomerControllerTest             1 sec 695 ms
```

Figure 7. The result of all controllers' tests

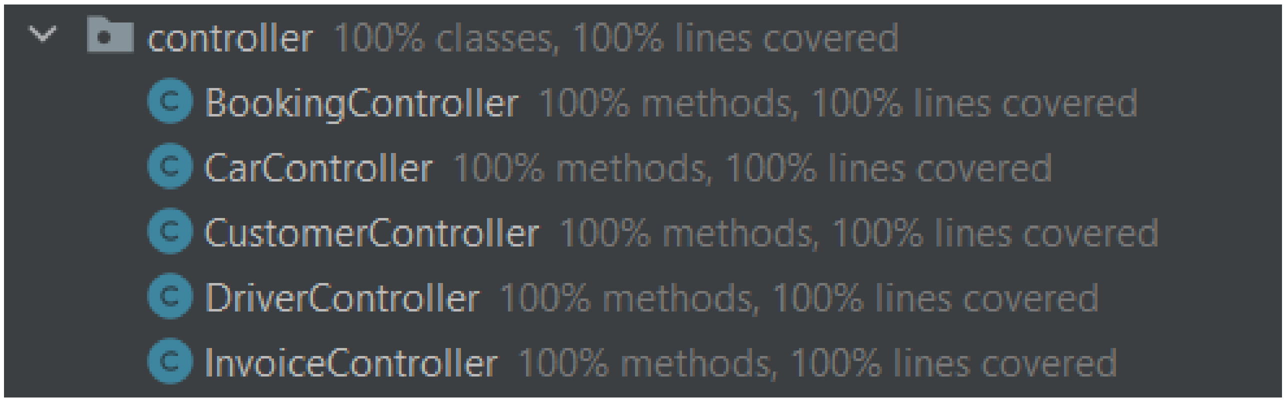

controller 100% classes, 100% lines covered
BookingController 100% methods, 100% lines covered
CarController 100% methods, 100% lines covered
CustomerController 100% methods, 100% lines covered
DriverController 100% methods, 100% lines covered
InvoiceController 100% methods, 100% lines covered

Figure 8. Controller classes with 100% coverage

LIMITATION & KNOWN BUGS

No validation of the date time format in request body

Although there is the validation process of the date time format in HTTP request parameters, the date time in request body of HTTP request is not validated but return a “Bad Request” status to the client side. This is due to not implementing a custom handler exception for the controllers to provide more meaningful and insightful warnings. However, we can resolve this problem in the future by creating an exception handler to interpret the invalid input more thoroughly to the client side.

Implementing LocalDatetime instead of ZoneDatetime

Because we apply LocalDatetime for pick – up and drop – off date time attributes of the Booking entity, our system may have some issues if it is used in different time zones in the world. We can repair this issue by applying ZoneDatetime to the attributes of date time instead so the date time can be determined more accurately and consistently according to different time zones in the world.

APPENDIX

1. **Usecase diagram PDF:** attached on ZIP folder (UML directory)
2. **Class diagram PDF:** attached on ZIP folder (UML directory)
3. **Github:** https://github.com/TuanDao2002/assignment2_cosc2440

REFERENCES

[1] D. Li and Z. Pang, "Dynamic booking control for car rental revenue management: A decomposition approach", *European Journal of Operational Research*, vol. 256, no. 3, pp. 850-867, 2017. Available: 10.1016/j.ejor.2016.06.044.

[2] R. F. Grove and E. Ozkan. "THE MVC-WEB DESIGN PATTERN ". scitepress.org. https://www.scitepress.org/Papers/2011/32969/32969.pdf (Accessed May 8th, 2022)

[3] S. Burbeck. " Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)". dgp.toronto.edu. http://www.dgp.toronto.edu/~dwigdor/teaching/csc2524/2012_F/papers/mvc.pdf (Accessed May 8th, 2022)

[4] P. Lalanda. " Shared repository pattern". citeseerx.ist.psu.edu. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.62.1879&rep=rep1&type=pdf (Accessed May 8th, 2022)

[5] Java Design Patterns. "Service Layer". java-design-patterns.com. https://java-design-patterns.com/patterns/service-layer/ (Accessed May 8th, 2022)