OBJECT ORIENTED PROGRAMMING TECHNICAL DOCUMENTATION

SpaceTravelApp

Martin Sivák

Faculty of Informatics and Informational Technologies
Slovak University of Technology
Bratislava, Slovakia

Name: Martin Sivák
AIS ID: 116291

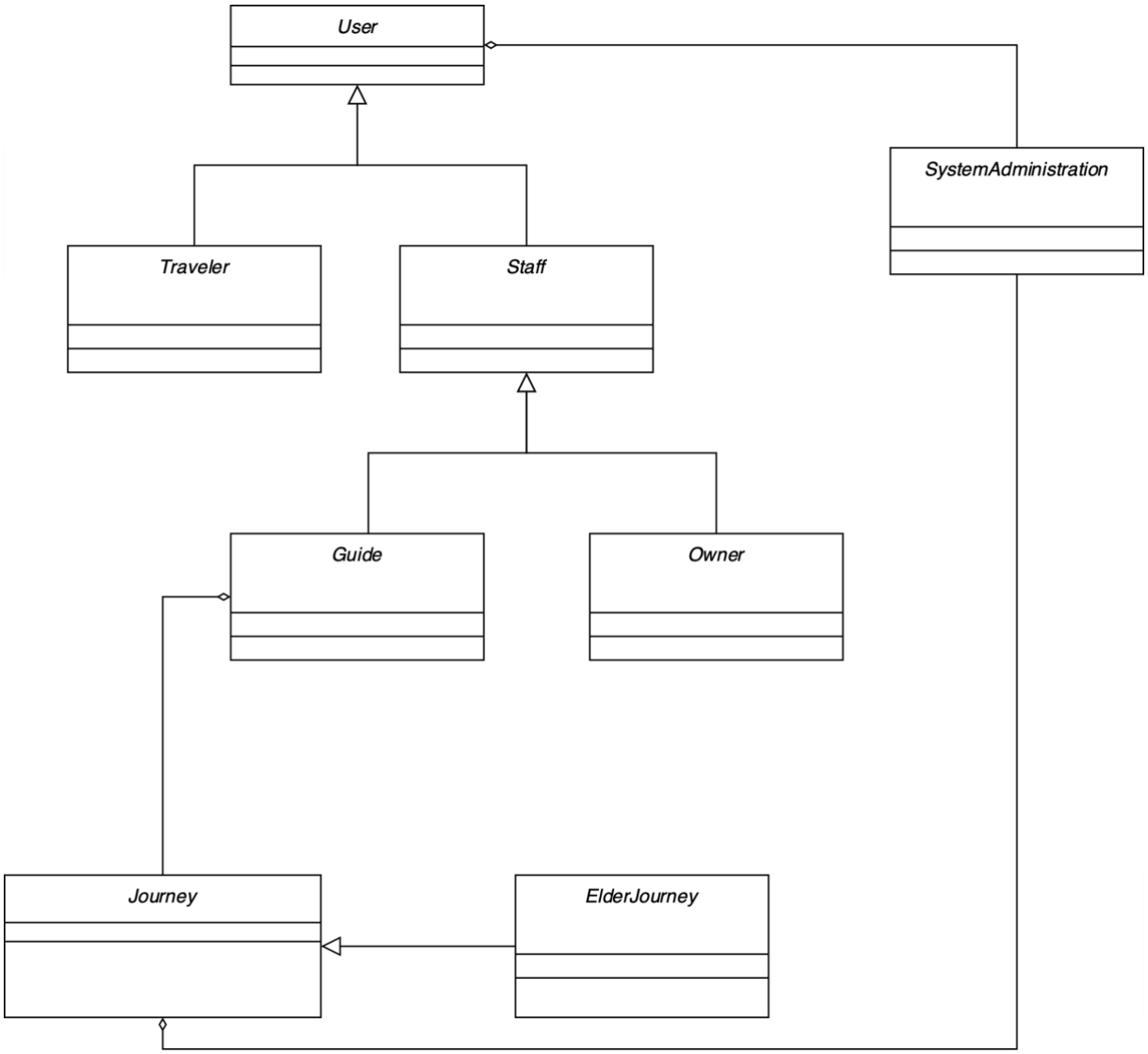
Contents

Introduction

The goal of this project was to create an app that would enable the management of future space travel. In our simple model app, we have three types of users: Traveller, Guide and the Owner. Each has its specific rights, but all share some common traits (like being able to deposit or withdraw money). Guides can create new journeys – either regular journeys or journeys where there is some age constrain (e.g., only user who is above certain age can purchase ticket to this journey). After journeys are created, the Owner user type must either approve them or cancel them – only approved journeys are displayed for the Travellers so they can be assigned to them. Travellers can add a journey to their journeys by paying for it – if they have enough money of course. The money Travellers paid is then assigned to the Owner who created the journey. If they decide that they do not want to participate in this journey anymore, they can cancel their participation on it, but since in our up we have non-refundable policy, they will not get their money back.

Name: Martin Sivák
AIS ID: 116291

Diagram



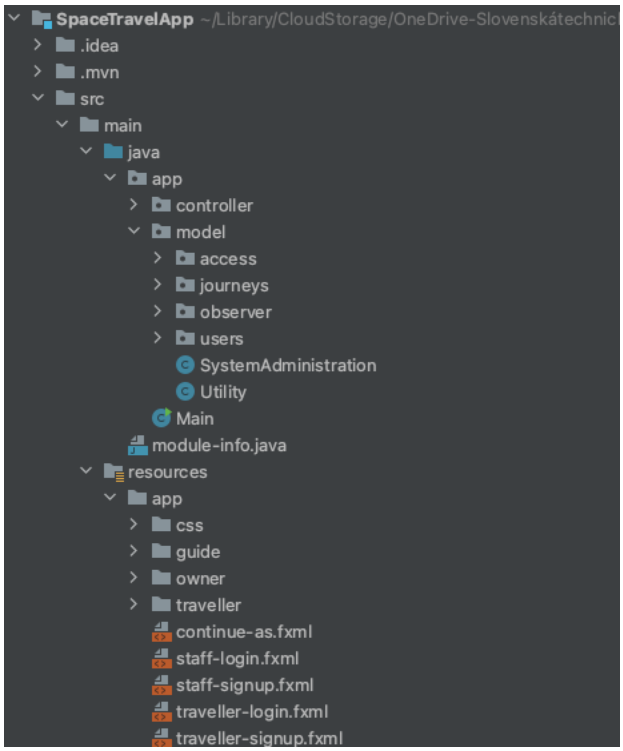
Name: Martin Sivák
AIS ID: 116291

Criterion Fulfilment

Mandatory Criterions

Code Organisation and MVC

Our code is organised into packages with logical structure and utilizing the MVC pattern, as can be seen below.



Encapsulation

We use encapsulation in every class. In the example below is the Journey class (*src – main – java – app – model -journeys*). We can see that all class fields have private access modifiers.

```
import java.util.ArrayList;

The top-level Journey class Every Journey is being observed by users and users are updated each time it
changes state
1 inheritor
public class Journey implements Observable {
    private long id;  6 usages
    private String name;  4 usages
    private int price;  4 usages
    private int loggedUsers;  9 usages
    private int capacity;  6 usages
    private Guide author;  3 usages
    private String date;  4 usages
    private JourneyState journeyState;  5 usages

    private ArrayList<Notifiable> observers;  4 usages
```

Name: Martin Sivák
AIS ID: 116291

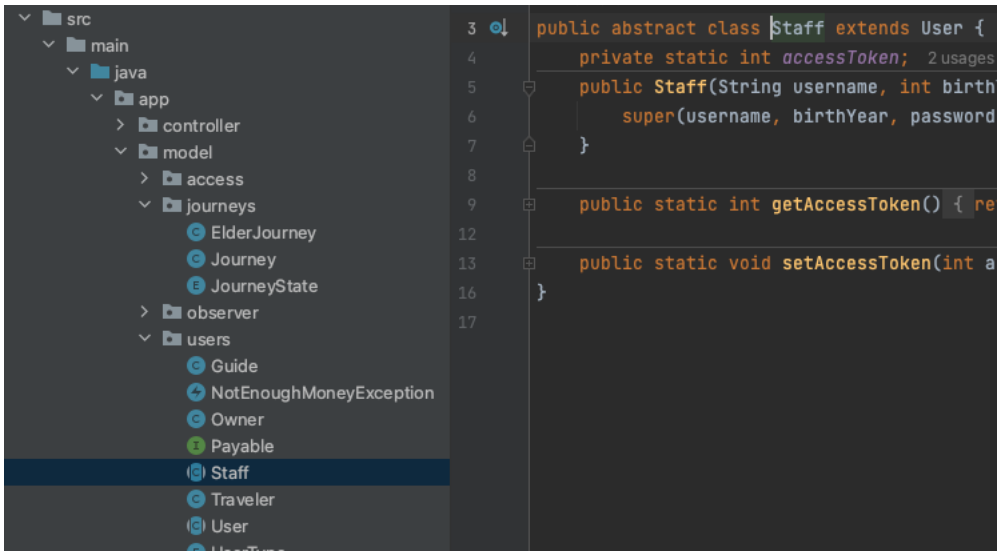
Aggregation

In this example in the class Guide (*src – main – java – app – model – users*) we create a new Journey. New Journey can come into existence only if the Guide object creates it and is bounded to it throughout its whole “life”.

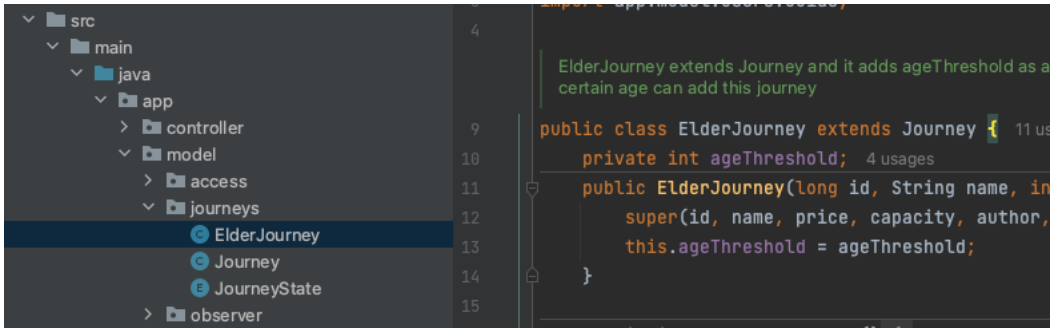
```
71
        Params: journeys – the list of existing Journeys
        Returns: a new Journey which attributes are generated
76      public Journey createJourney(ArrayList<Journey> journeys) {...}
```

Inheritance

The User class is the top-level class, from it the classes Traveller and Staff inherit some properties, and from Staff inherit the classes Guide and Owner.



In the second hierarchy, the ElderJourney class inherits from the Journey class.



Name: Martin Sivák
AIS ID: 116291

Polymorphism

In the class Journey (*src – main – java – app – model -journeys*) we call method updateObservers each time the journey object changes its JourneyState property (more on that in the Observer Design Patter chapter). In the updateObservers method we iterate over a list of observers and call the method update on all of them. In our project observer can be any object from the User class (Traveller, Guide, Owner) or more formally any class that implements the Notifiable interface. I each class the method update is implemented in different way.

```
@Override  2 usages
public void updateObservers() {
    for (Notifiable observer : observers) {
        observer.update( subject: this);
    }
}
```

```
@Override  1 usage
public void update(Observable subject) {...}
```

Method Overloading

Overloaded methods generateRandom from the Utility class (*src – main – java – app – model*).

```
public static long generateRandom(long min, long max) {  9 usages
    if (min > max) {
        long temp = min;
        min = max;
        max = temp;
    }

    return (long) (Math.random() * (max-min) + min);
}

public static double generateRandom(double min, double max) {  no usages
    if (min > max) {
        double temp = min;
        min = max;
        max = temp;
    }

    return (Math.random() * (max-min) + min);
}
```

Name: Martin Sivák
AIS ID: 116291

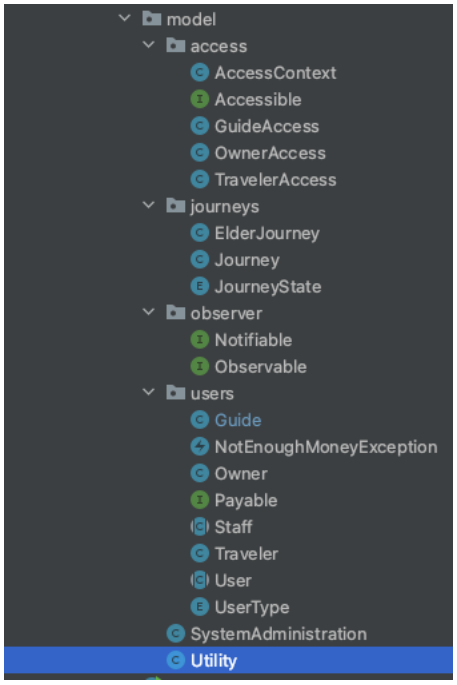
Method Overriding

The classes Traveller, Guide, and Owner each provides its own implementation of the update method.

```
@Override  1 usage
public void update(Observable subject) {...}
```

Interfaces

In our model package, we have four interfaces: Accessible, Notifiable, Observable, and Payable.



Here is the code for the Payable interface. This interface provides users with methods for transferring, increasing, and decreasing amount of money on their TravelCard.

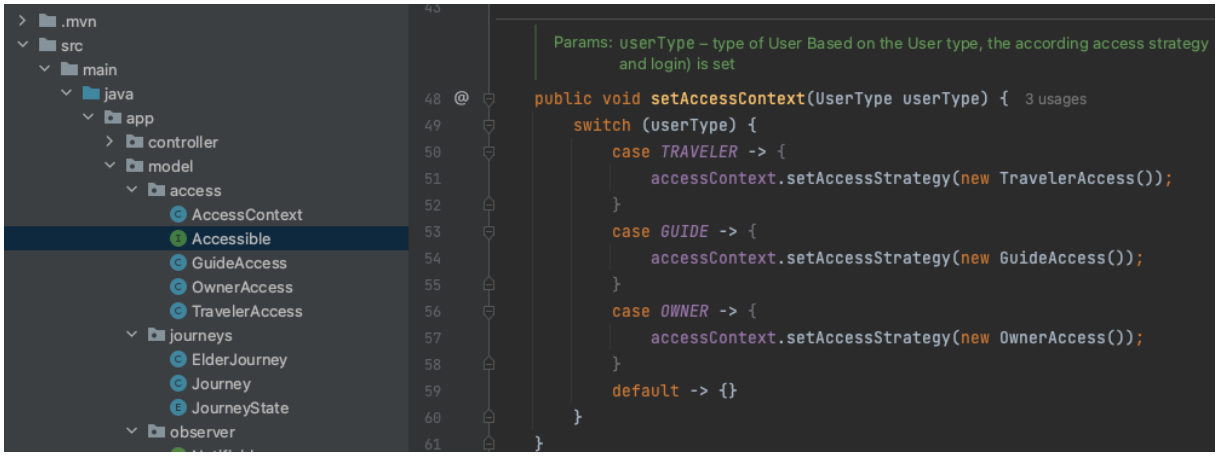
```
package app.model.users;


Payable interface is implemented by all Users
5 implementations
public interface Payable {  1 usage
    3 implementations
    boolean pay(double amount, User to);  1 usage
    3 implementations
    boolean withdraw(double amount);
    3 implementations
    boolean deposit(double amount);
}
```

Name: Martin Sivák
AIS ID: 116291

Other Criterions

Strategy Design Patter

All classes and interfaces for the Strategy design pattern are in the access package. Here we have three different strategies how we can sign up or login the user, based on the user type. After the user picks as who he wants to continue into the application, the according access strategy is choose by setting the accessContext. The code bellow is from the SystemAdministration class where we are setting the context according to a user type.



Observer Design Patter

We find the Observer design pattern as a perfect match in our case. Interfaces for the Observer DP are in the package observer. Basically our users are observers that observe the journeys and are notified (or **updated**) every time a journey changes state.

We keep references to all users and all journeys in the SystemAdministration class. When new journey is added, we call the method setObservers so that all users start “observing” the journey. On the other hand, if a new user signs up for our application, we call the method setSubjects – the users starts “observing” all existing journeys.

Name: Martin Sivák
AIS ID: 116291

```
Params: subject – journey that was recently added to the system After the new journey is added to
        the system, it updates all user about its state
private void setObservers(Observable subject) {  1 usage
    for (User user : users){
        subject.addObserver(user);
    }
}

/**
 * @param observer user that was recently added to the system
 *                 After the new user is added to the system, it is updated by all journeys about their state
 */
private void setSubjects(Notifiable observer) {  1 usage
    for (Journey journey : journeys){
        journey.addObserver(observer);
        journey.updateObservers();
    }
}
}
```

Exception – Throw and Catch

We have created a new Exception class called NotEnoughMoneyException.

```
package app.model.users;

/**
 * Exception that is thrown after a user tries to pay, but he has not enough money on its TravelCard
 */
public class NotEnoughMoneyException extends Exception {  9 usages

    public NotEnoughMoneyException(String message) { super(message); }
}
```

We throw this exception in the nested class of the User class – the TravelCard class, when user tries do decrease the amount of money, he has, by a greater amount than he has.

```
67
68      public boolean decreaseBalance(double by) throws NotEnoughMoneyException {  7 usages
69          if (this.balance - by < 0 || by <= 0) {
70              throw new NotEnoughMoneyException("Cannot withdraw more money than you own.");
71          }
72          this.balance -= by;
73          return true;
74      }
75  }
```

And we catch the exception for example in the Traveller class, where Traveller first **tries** to pay some amount of money, and if he has not that amount of money, the exception is thrown and caught and handled appropriately. We can see the effect of this, when Traveller in the

Name: Martin Sivák
AIS ID: 116291

application tries to add a journey which costs more that the actual balance Traveller has – the pop-up window is displayed informing us about the problem and possible solution.

```java
@Override  1 usage
public boolean pay(double amount, User to) {
    try {
        getTravelCard().decreaseBalance(amount);
    } catch (NotEnoughMoneyException exception) {
        System.out.println("Not enough money on card.");
        return false;
    }

    return to.getTravelCard().increaseBalance(amount);
}
```

GUI

This application comes with a GUI. More on the GUI and an example user flow can be seen in the Appendix A.

Multithreading

After user is successfully logged into his account, we start a new Timer object (which starts a new task on separate thread) that starts measuring the duration of user's session – we can see this in the application. The code below is from the SystemAdministration class.

```java
    private void startSession() {  2 usages
        timer.schedule(() -> { sessionDuration++; },  delay: 0,  period: 1000);
    }

    private void endSession() {  1 usage
        timer.cancel();
        sessionDuration = 0;
        timer = new Timer();
    }
```

```java
public int login(String username, int password, int token) {
    int responseCode = accessContext.login(username, password, users, token);
    if (responseCode == 0) {
        setCurrentUser(username);
        startSession();
    }

    return responseCode;
}

Signs out currently logged User, ends his session and prints all users to the console
public void logout() {  3 usages
    currentUser = null;
    endSession();
    printUsers();
}
```

Name: Martin Sivák
AIS ID: 116291

RTTI

One of the RTTI examples in the Guide class where we call the update method and check whether the subject object is from the Journey or ElderJourney class and react accordingly.

```java
@Override  1 usage
public void update(Observable subject) {
    if (subject.getClass() != Journey.class && subject.getClass() != ElderJourney.class ) {
        return;
    }

    Journey journey = (Journey) subject;
    JourneyState journeyState = journey.getJourneyState();

    if (journeyState == JourneyState.PENDING && journey.getAuthor() == this) {
        if (myJourneys.contains(journey)) {
            return;
        }
        myJourneys.add(journey);
        return;
    }

    if (journeyState == JourneyState.CANCELLED && journey.getAuthor() == this) {
        if (canceledJourneys.contains(journey)) {
            return;
        }
        canceledJourneys.add(journey);
        return;
    }

    if (journeyState == JourneyState.AVAILABLE && journey.getAuthor() == this) {
        if (approvedJourneys.contains(journey)) {
            return;
        }
        approvedJourneys.add(journey);
    }
}
```

Another example is from the controller package, class called Signup, method onNextButtonClick, where we capture the current User object. After that, based on the child class we decide what screen we want to show.


```java
switch (responseCode) {
    case 0 -> {
        // switch scene based on user type
        User currentUser = systemAdministration.getCurrentUser();
        if (currentUser.getClass() == Traveler.class) {
            ControllerUtility.switchSceneOnEvent(event, PathToFXML: "traveller/traveller-main.fxml");
        } else if (currentUser.getClass() == Guide.class) {
            ControllerUtility.switchSceneOnEvent(event, PathToFXML: "guide/guide-main.fxml");
        } else {
            ControllerUtility.switchSceneOnEvent(event, PathToFXML: "owner/owner-main.fxml");
        }
        clear();
    }
    case 1 -> {
        setErrorMessage("User with name \"" + username + "\" already exists.");
    }
    case 2 -> {
        setErrorMessage("Should not get this code");
    }
    case 3 -> {
        setErrorMessage("Incorrect access token for user with name \"" + username + "\".");
    }
```

Name: Martin Sivák
AIS ID: 116291

Nested Class

In the User class we have a nested TravelCard class.

```
4 inheritors
public abstract class User implements Payable, Notifiable {
    private String username;  2 usages
    private int password;  2 usages
    private TravelCard travelCard;  3 usages
    private final UserType currentUserView;  2 usages
    public User(String username, int birthYear, int password, UserType currentUserView) {
        this.username = username;
        this.password = password;
        this.travelCard = new TravelCard(
                Utility.generateRandom(1_000_000_000, 10_000_000),
                birthYear,
                balance: 0
        );
        this.currentUserView = currentUserView;
    }

    /**
     * Each user has its own TravelCard with unique number
     */
    public class TravelCard {  3 usages
        private String name;  2 usages
        private long number;  2 usages
        private int birthYear;  2 usages
        private double balance;  5 usages

        public TravelCard(long number, int birthYear, int balance){  1 usage
            this.number = number;
            this.birthYear = birthYear;
            this.balance = balance;
        }
```

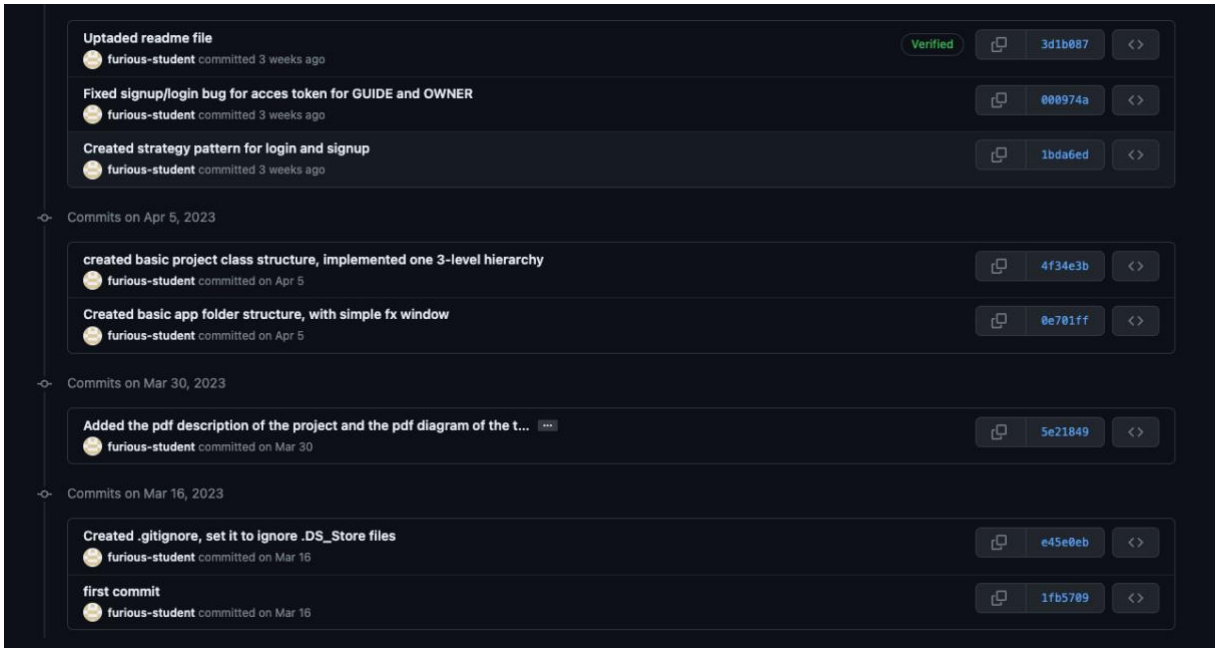
Name: Martin Sivák
AIS ID: 116291

Main Versions

We would divide the commits into three main parts:

First

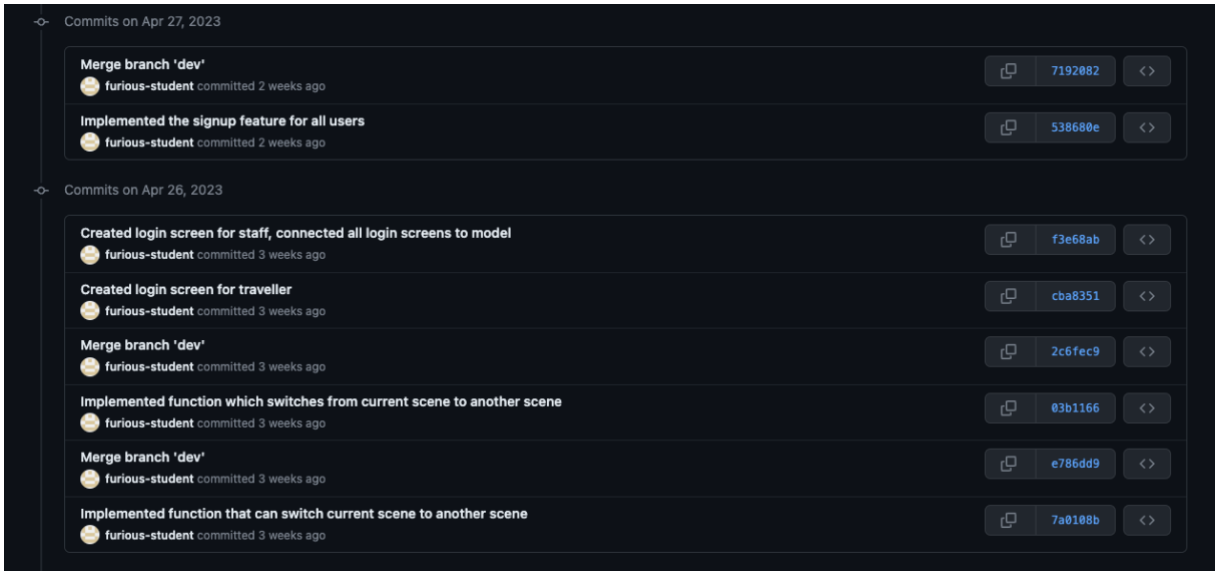
Here we created the basic layout of the project, folder structure and implemented the first inheritance hierarchy – Users. We also implemented the login and signup functionality and the strategy design patter.



Second

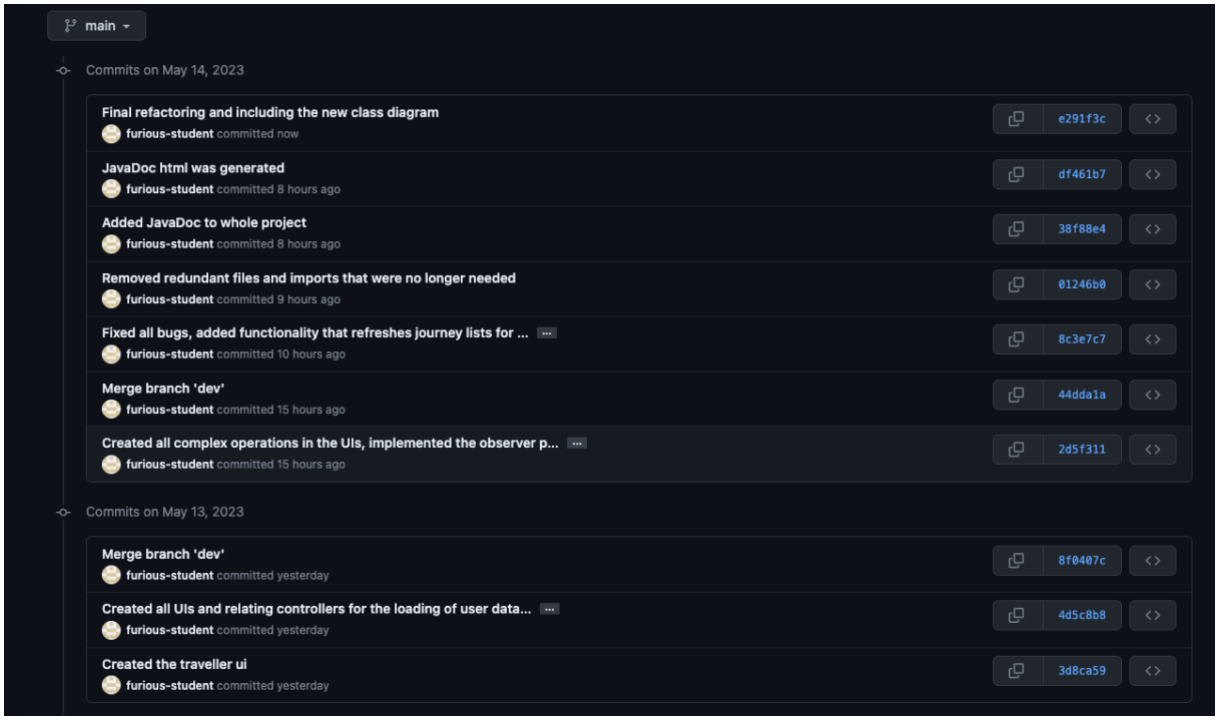
In the second part we started implementing and styling the GUI for our app – we created the login and screens signup screens and connected it to our model part via the controller. Users could now log into the application but could not do anything else.

Name: Martin Sivák
AIS ID: 116291



Third

In the third and final part, we implemented the rest of the UI – screens for all users. We implemented the second inheritance hierarchy – the Journeys. We also implemented the Observer design pattern and defined the operations each user type can perform on the journeys. After completing our implementation we commented our code with the JavaDoc comments and generated the corresponding HTML output. In the end we refactored our code, removed unnecessary imports, comments, print statements etc.

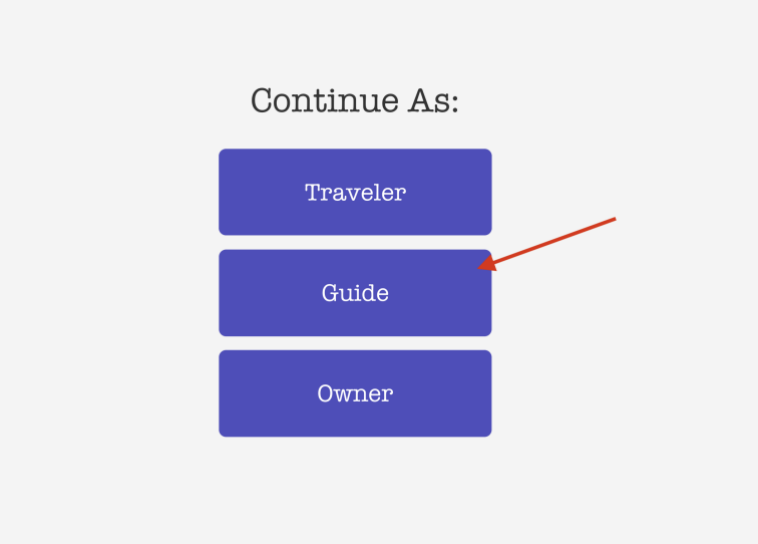


Name: Martin Sivák
AIS ID: 116291

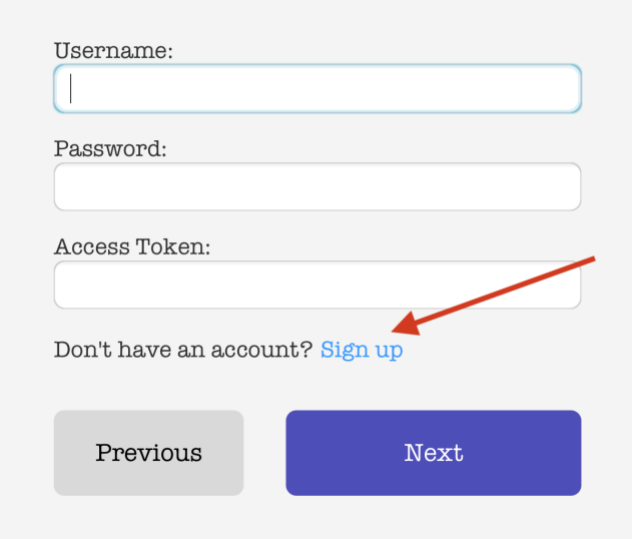
Appendix A – Example User Flow

In this section we walk you through a simple usage of our application. Be aware that you can use the application as you wish, we just want to make it more clear to you how it works so you become more familiar and confident while using it.

1. After the start, you should see the following screen asking you to choose a type of user that you want to continue as. Select the Guide.

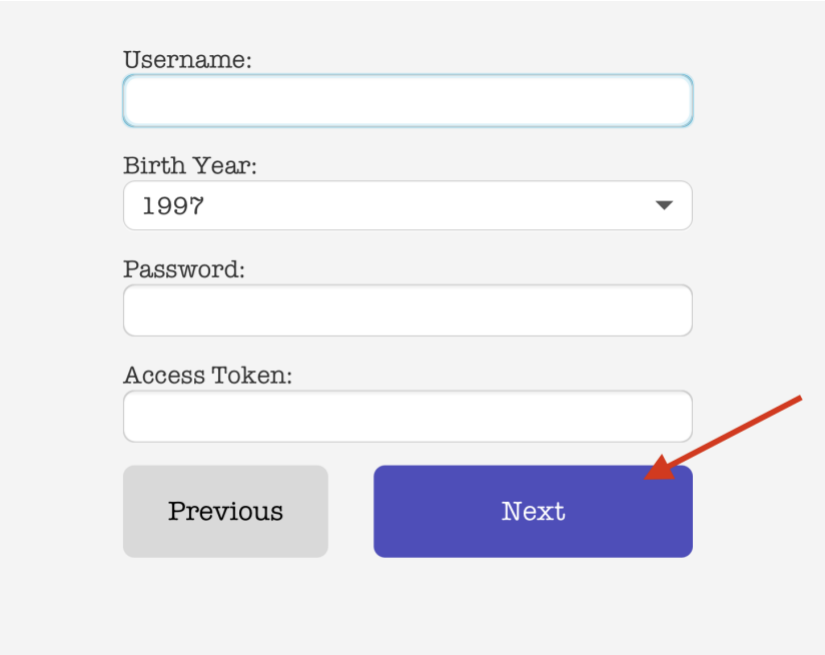


2. You will be asked to log in, but you do not have an account yet. Click on the sign up link.

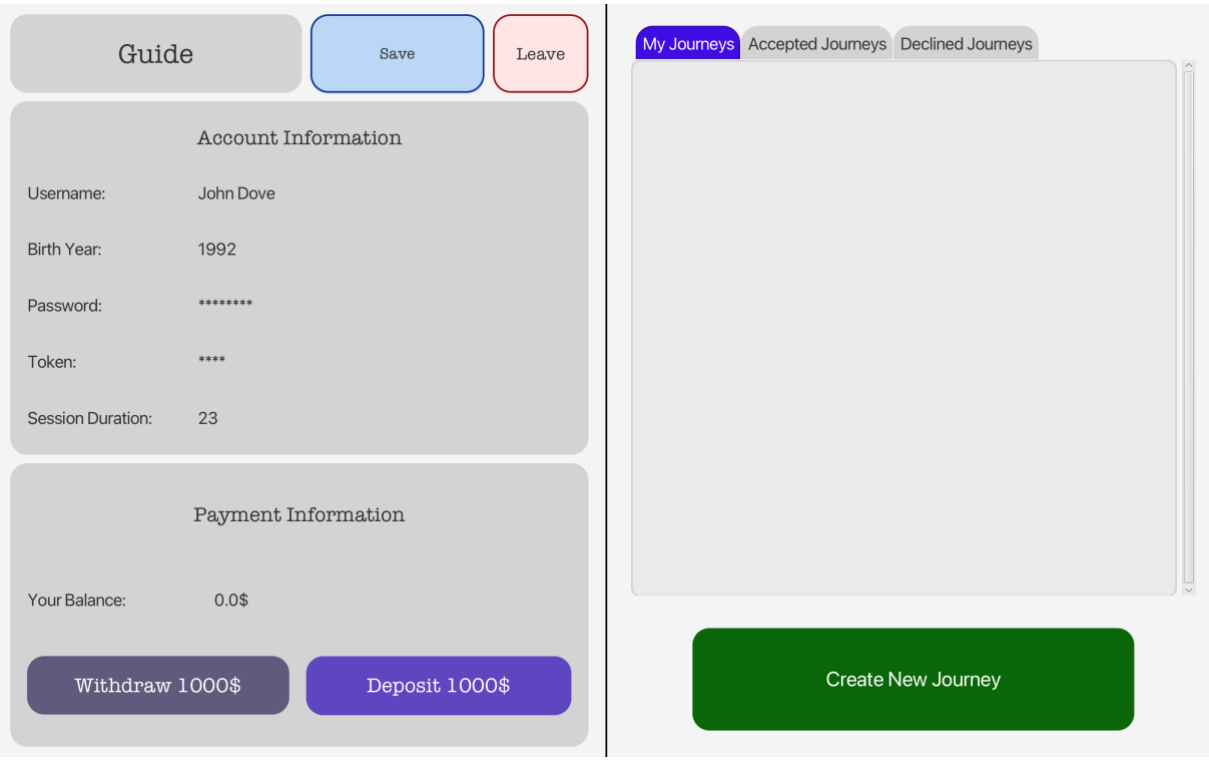


Name: Martin Sivák
AIS ID: 116291

3. Fill the fields and click the next button while keeping these things in mind:
    a. The username must be at least 3 characters long.
    b. The password must be at least 8 characters long, must contain only numbers and cannot start with number 0.
    c. Access token for Guide is always 1111



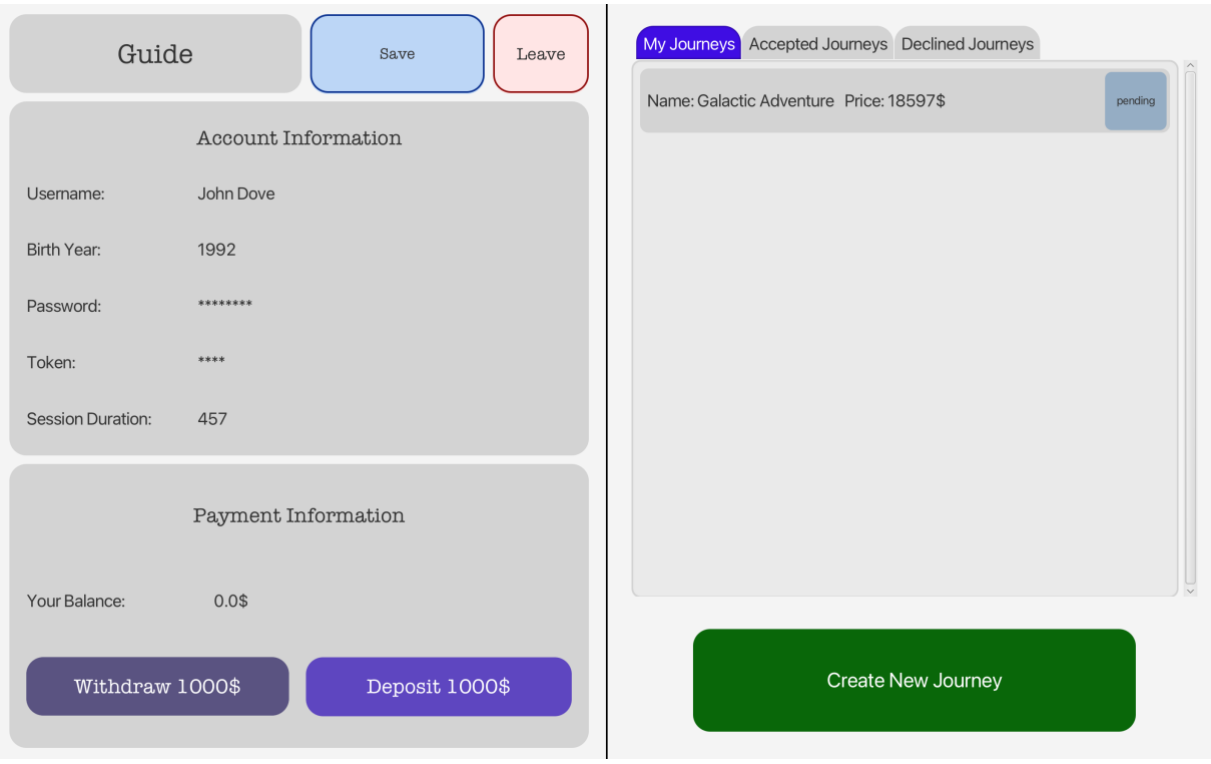
4. You should see the following screen:



Quick description:

- The save button doesn't do anything yet.
- The Leave button will log you out from the application.
- On the left side you can see your information.
- In the Payment Information section you can add or deposit money (each click 1000$). Feel free to try it!
- On the right side you can see a tab pane with three tabs:
  - **My Journeys**: Here you can see all journeys created by you.
  - **Accepted Journeys**: Here you can see you journeys that were accepted by some owner.
  - **Declined Journeys**: Here you can see you journeys that were declined by some owner.
- With the button Create New Journey you can create a brand-new Journey. Do it. Pop-up will appear, informing you that the new journey was created.

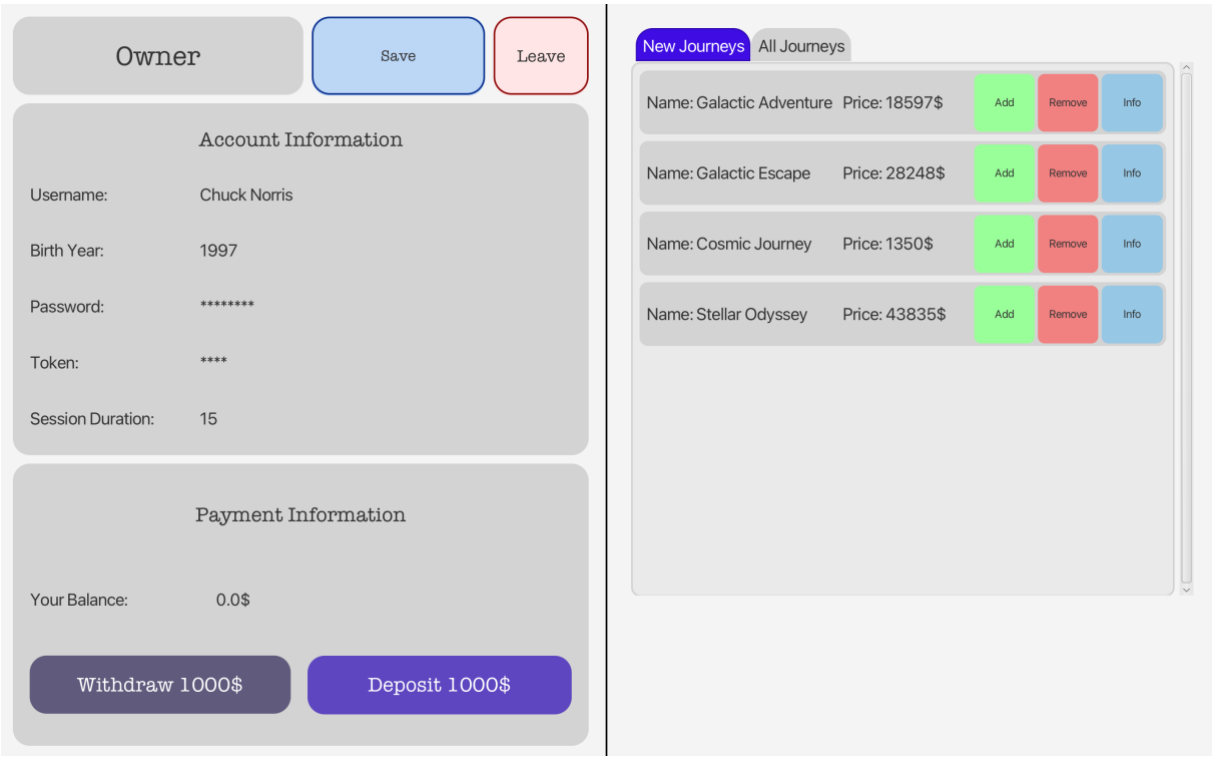
5. Close the pop-up. You should see the new journey in the "My Journeys" tab with the label "pending".



6. Feel free to create any number of journeys but create at least four. Now we need to log in as an Owner and manage the new journeys.
7. Log out of the application by clicking the Leave button on the top.
8. You should be able to see the initial screen.

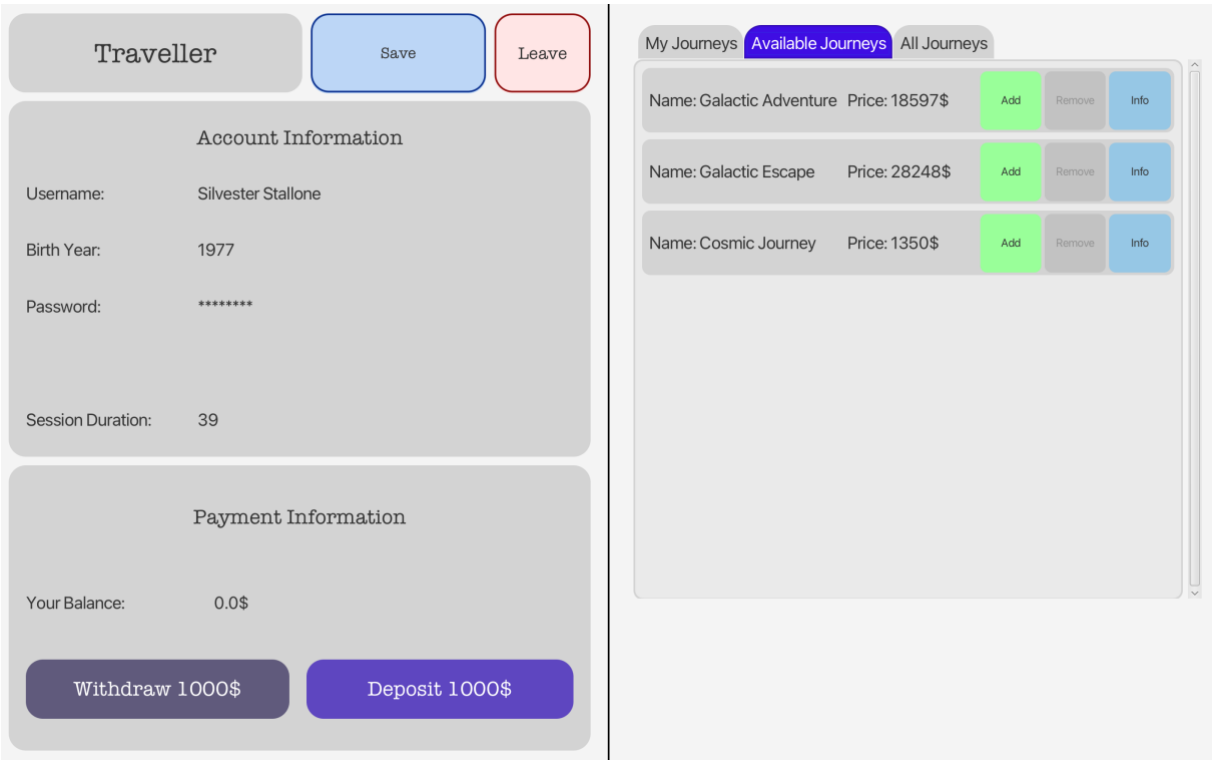
Name: Martin Sivák
AIS ID: 116291

9. Choose now that you want to log in as Owner.
10. Repeat the signup process in the same way we did. All constrains for the input fields stay the same except that the token for the guide is 9999. Log into the application. You should see following screen:



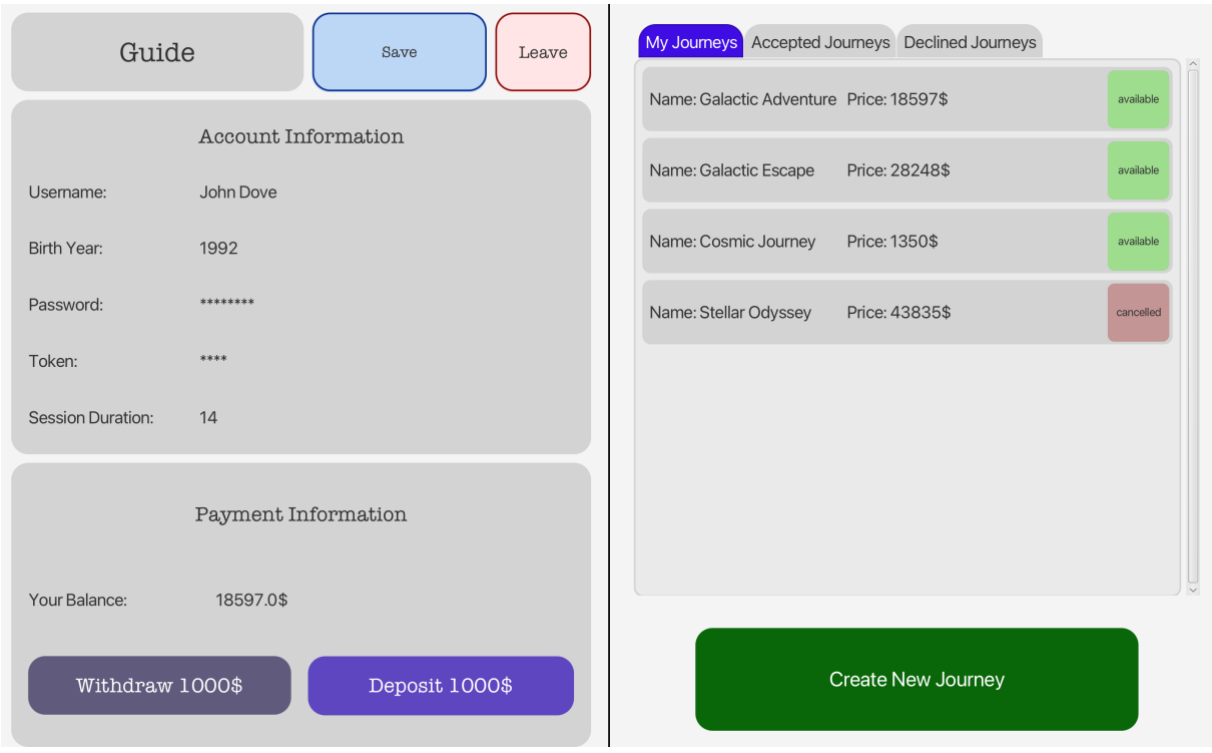
11. The left side of the screen works similarly to the Guide's. The interesting part is on the right. Again, you can see the tab pane with two tabs:
    a. **New Journeys**: Here are all new journeys created by the Guides, where decision needs to be made.
    b. **All Journeys**: Here are all Journeys that were either accepted or declined. You can only see their info but cannot rethink your decision and change their state again.
12. Try to accept three journey and decline one. You can see that they are added to the "All Journeys" tab and removed from the "New Journeys" tab. Then log out of the application.
13. Now sign in as a Traveller. Same steps like before, but you do not have to input any token. You shall see this screen:

Name: Martin Sivák
AIS ID: 116291



14. Notice that the “My Journeys” tab is empty. That’s because you did not assign for any journey yet! So, let’s do this.
15. Click on the “Available Journeys” tab. It is possible that the number of available journey there will be lower than on the example screen, because if there is any age restriction for a journey that you do not fulfil, it won’t be displayed in the “Available Journeys” tab. To see all journeys, click on the “All Journeys” tab.
16. Now if you try to add a journey, you will get an error pop-up message saying that you do not have enough money to be assigned to that journey. Click ok to close it.
17. Increase the amount of money you have by clicking the deposit button. Deposit enough money to be able to buy at least one journey.
18. Click the add button on a journey you want to add. Notice that it will be added to your journey, and you can only remove it or see the info about it. Also notice that your balance was adjusted.
19. Log out of the application.
20. Log in as a Guide with the account we used to sign up:

Name: Martin Sivák
AIS ID: 116291



21. Few things to notice here:
    a. Notice that the state of your journeys was changed: they are either available or cancelled. You can also filter the by clicking on the accepted journeys or cancelled journeys tab.
    b. The balance on your account has been increased by the cost of the journey.
22. And that's it! Now you know everything for navigating freely and fearlessly in the application.

You can create new user accounts, create journeys, change their states, withdraw or deposit money, etc.

Name: Martin Sivák
AIS ID: 116291

Conclusion

To conclude this project, we think that it was extremely exhausting and sometimes frustrating to implement a larger application but also on the other hand truly engaging and fun activity that helped us understand the Java programming language and the object-oriented principles on a deeper level, solidified our knowledge and increased our curiosity to learn more about these topics.