Gotcha Scooter Mobileye ADD

Amit Moskovitz, Tom Nisim, Alexander Uzelevesky

Table of content

Use Cases

Use Case 1 : Initialization

Use-case 1.a: Server Initialization

- **Actor:** Admin
- **Precondition:** None
- **Parameters:** Configuration File Parameters
- **Actions:**
    - Admin launches server app
    - default admin is set
    - default values from Configuration File Parameters are set.

Action	Data	Expected Result
Initialize the System	The configuration file parameters	A system Admin has been set (At least one) and default values from the configuration file were loaded.

Use-case 1.b: Raspberry Pi Initialization

- **Actor:** Rider
- **Precondition:** None
- **Parameters:** Configuration File Parameters
- **Actions:**
    - Rider initializes raspberry pi device
    - Raspberry Pi reads data from configuration file
    - Raspberry Pi updates his internal data from the data received from the configuration file

Action	Data	Expected Result
Initialize the System	The configuration file parameters	Raspberry Pi data is updated according to the configuration file

Use-case 2: Change External Service

- **Actor:** Admin
- **Precondition:** Admin is logged in
- **Parameters:** New External Service
- **Actions:**
    - Admin will choose different external services from the list of available services.
    - The system will verify the service is a valid choice.
    - The system will update the Server to use this service.

Action	Data	Expected Result
Change External Service	Valid external service	The system will be updated to use the new external service.
	Invalid External service	The system will alert the Admin that the service is invalid.

Use-case 3: Register new User

- **Actor:** Rider
- **Precondition:** None
- **Parameters:** Email, Password, gender, name, last_name, rp_serial_number,birthday, licence_issue_date
- **Actions:**
    - Rider will enter his credentials.
    - The Server will verify the Email is not being used by another user.
    - If the email is being used already it will alert.
    - The Server will verify the password is secure.
    - The Server stem will add the new user to the list of active users.
    - The Server will create a mapping between the rp_serial_number and the user email

Action	Data	Expected Result
Add New User	Valid Email and password	The Server will add the new user to the list of active users.
	Invalid Password	The Application will alert the User that the password is invalid and will not register the new user
	Invalid Email	The Application will alert the User that the Email is invalid and will not register the new user

Use-case 4: Edit User Profile

- **Actor:** Rider
- **Precondition:** Rider is an Active user
- **Parameters:** Age, Gender, type of Scooter, Driver's license issue date
- **Actions:**
    - The Rider will enter his new desired profile information.
    - The Application will verify the validity of the information.
    - The Application will send the new information to the server which will Update the user Profile.

Action	Data	Expected Result
Edit User Profile	Valid new information	The system will update the information on the user profile.
	Invalid Age	The Application will alert the user that the Age is invalid
	Invalid type of scooter	The Application will alert the user that the type of scooter is invalid
	Invalid Driver's license issue date	The Application will alert the user that the Driver's license issue date is invalid

Use-case 5: User Login

- **Actor:** Rider
- **Precondition:** None
- **Parameters:** Email, Password
- **Actions:**
    - The Rider will enter his credentials.
    - The Application will send the new information to the server which will validate the information.
    - If the credentials are valid the user will be logged on, otherwise he will be alerted.

Action	Data	Expected Result
User Login	Valid user credentials	The User will be logged in.
	Incorrect Password	The Application will alert the User that the password he provided is Incorrect .
	Incorrect Email	The Application will alert the User that the Email he provided is Incorrect or not existing

Use-case 6: User Logout

- **Actor:** Rider
- **Precondition:** Rider is logged in
- **Parameters:** None
- **Actions:**
    - The Rider will press the logout button.
    - The Application will send the new information to the server which will mark the scooter driver as logged out.

Action	Data	Expected Result
User Logout	None	The User will be logged out.

Use-case 7: Send Message to Admins

- **Actors:** Rider
- **Precondition:** Rider is logged in
- **Parameters:** Title, Description
- **Actions:**
    - The Rider will enter the title and description of his message.
    - The Application will verify the title/ description don't contain malicious data and that they are not empty.
    - The application will send the message to the server.
    - The Server will alert the Admin about the message.

Action	Data	Expected Result
Send Message To Admins	Valid title and description.	The Application will send the message to the Server which will send the message to the appropriate Admin
	Invalid title	The Application will alert the User that the title is invalid
	Invalid description	The Application will alert the User that the description is invalid

Use Case 8 : Start of Riding

Use-case 8.1: Suggest safest routes

- **Actors:** Rider
- **Precondition:** Rider is logged in, system is connecting to maps external system
- **Parameters:** destination address
- **Actions:**
    - rider will start a ride using the rider app
    - The system will get the user location via phone GPS
    - The system will send to the External Maps Service the destination address and the driver location
    - The External Maps Service will send to the system a list of routes from the driver location to the destination address.
    - The system will rate each route according to its level of safety according to defined indicators
    - The system will show the driver the safest routes

Action	Data	Expected Result
Suggest Safest routes	Destination address does not exist	The system will show to the user a specific error message
	Legal destination address	The System will show the driver the possible safest routes.

Use Case 9 : During the ride

Use-case 9.1: receiving road video shooting

- **Actor:**
- **Precondition:** The Raspberry pi is connected to a camera
- **Parameters:** None
- **Actions:**
    - The Raspberry Pi gets the video shooting from the camera
    - The Raspberry Pi check the clarity of the video
    - If the video shooting is unclear , the Raspberry Pi gets another road video shooting from the camera

Receiving road video shooting	Clear video shooting of the road	The Raspberry Pi perform Use Case 9.1.2
	Unclear video shooting of the road	The Raspberry Pi gets another road video shooting from the camera

Use-case 9.2: Identify stationary objects

- **Actor:**
- **Precondition:** A deep learning process of the system for identifying stationary objects according to a collection of images of such objects is done.
- **Parameters:** Road video shooting(Use Case 9.1)
- **Actions:**
    - The Raspberry Pi analyze the video to detect stationary objects in the rider path - Potholes in the road, trees, electricity poles
    - The system calculate the distance between the rider and the object, and the scooter's current speed
    - When the rider reaches the location where he is supposed to receive an alert (according to the previous section and predetermined parameters), the system creates an alert

Identify stationary objects	The video shooting of the road which does not contain stationary objects in the rider's path.	No alert tone will be heard.
	The video recording of the road which contains stationary objects in the rider's path.	An alert tone will sound for each stationary object detected in the path.

Use-case 9.3: Identify portable objects

- **Actor:**
- **Precondition:** A deep learning process of the system for identifying stationary objects according to a collection of images of such objects is done.
- **Parameters:** Road video shooting(Use Case 9.1)
- **Actions:**
    - The Raspberry Pi analyzes the video to detect portable objects in the rider path – Pedestrians or cars. The identification is done according to the deep learning process that was carried out (precondition).
    - The system calculates the distance between the rider and the object, the scooter's current speed, and the speed and direction of movement of the portable object.
    - When the rider reaches the location where he is supposed to receive an alert (according to the previous section and predetermined parameters), The system creates an alert.

Action	Data	Expected Result
Identify portable objects	The video shooting of the road which does not contain portable objects in the rider's path.	No alert tone will be heard.
	The video recording of the road which contains portable objects in the rider's path.	An alert tone will sound for each portable object detected in the path.

Use-case 9.4: Identify Type of riding surface

- **Actor:**
- **Precondition:** A deep learning process of the system for identifying road surface type according to a collection of road surface types and their classifications is done
- **Parameters:** Road surface data
- **Actions:**
    - The Raspberry Pi analyzes the road surface data to detect the type of the surface. The identification is done according to the deep learning process that was carried out (precondition).

Action	Data	Expected Result
Identify type of riding surface	Road surface data - roadway	The Raspberry Pi identify the type of road surface successfully
	Road surface data - sidewalk	The Raspberry Pi identify the type of road surface successfully
	Road surface data - other	The Raspberry Pi failed to identify the type of road surface

Use Case 10 : End of riding

Use-case 10.1: Sending data to the server

- **Actor:**
- **Precondition:** Use Cases 9.2-9.4
- **Parameters:**
- **Actions:**
    - The Raspberry Pi sends the data collected in Use Cases 9.2-9.4 to the server

Action	Data	Expected Result
Sending data to the server		Data sends to the server successfully

Use-case 10.2: Saving information about the locations of the dangers

- **Actor:**
- **Precondition:** connection with the DB, connecting to maps external system, detection of object mark as a hazard (Use Cases 9.2 - 9.3)
- **Parameters:**
- **Actions:**
    - The system gets the exact coordinates of the detected hazards from the ride(use case 9.2-9.3) by using maps external service API.
    - The system identifies the city which the hazard detects in by using maps external service API and by the coordinates collected in the previous section.
    - The system saves the information from the previous sections to the DB with the specific object ID.

Action	Data	Expected Result
Saving information about the locations of the dangers.		The coordinates and city which the object detects are saved to the DB with the specific object ID.

Use-case 10.3: Riding experience questionnaire

- **Actor:** User
- **Precondition:** connection with the DB, User is connected to the system, The system detected the end of the riding
- **Parameters:** None
- **Actions:**
    - The system displays to the user riding experience questionnaire.
    - The user can fill in the questionnaire or skip .
    - If The user can fill in the questionnaire ,the system saves the questionnaire results to the DB.

Action	Data	Expected Result
Riding experience questionnaire	Users taking the option of filling up the questionnaire.	The user fills up the questionnaire, The results are saved to the DB.
	User decided to skip the option of filling up the questionnaire.	The questionnaire will not be displayed to the user.

Use-case 10.4: Saving riding data

- **Actor:**
- **Precondition:** connection with the DB , Use Cases 9.2-9.4.
- **Parameters:** None
- **Actions:**
    - The system saves the collected riding data (Glossary) to the DB .

Action	Data	Expected Result
Saving riding data		Riding data saved to the DB successfully.

Use-case 11: View Riding history

- **Actor:** User
- **Precondition:** connection with the DB, User is connected to the system
- **Parameters:** None
- **Actions:**
    - The system retrieved from DB the data about the ridings taken by the connected user (Use Case 10.3).
    - The system shows the data from the previous section to the connected user.

Action	Data	Expected Result
View Riding history	None	Riding data showed to the user.
	None	User does not have any riding history, specific message shown to the user :"No Riding History To Show"

Use-case 12: Users rating

- **Actor:** System
- **Precondition:** Use Case 10.3
- **Parameters:** None
- **Actions:**
    - The system retrieved from DB the data about the ridings taken by the connected user (Use Case 15).
    - The system rates the registered user by the data from the previous section and by the terms from Appendix C.

Action	Data	Expected Result
Users rating		The system rates the users successfully, The ratings match the terms from appendix C.

Use-case 13: Delayed notifications

- **Actor:** Admin
- **Precondition:**None
- **Parameters:** Notification content, the recipient user details
- **Actions:**
    - The admin sends the notifications content to the recipient user.
    - If the user is connected to the system – he will get the notification right away.
    - Else, the notification will be displayed when logging in.

Action	Data	Expected Result
Users rating – connected User	Notification content, user email(User is not connected to the system)	The user gets the notification right away and can view the notification content.
User rating - Guest	Notification content, user email(User is connected to the system)	When logging in, the user gets the notification and can view its content.

Use-case 14: Admin view user's data - ridings, rating and profile

- **Actor:** Admin
- **Precondition:** Use Case 10.3, connection with the DB
- **Parameters:** User details, data type (ridings, ratings, profile)
- **Actions:**
    - The system retrieved from the DB the data type of the user.
    - The system shows the information from the previous section to the admin.

Action	Data	Expected Result
Admin view user's data - ridings, rating and profile	Valid user email,Valid data type	The admin gets the data type of the user successfully.
	Invalid user email,Invalid data type	Specific error message shown to the admin

Use-case 15: Manage User

Use-case 15.1 : Remove user

- **Actor:** Admin
- **Precondition:** User is registered to the system.
- **Parameters:** User email
- **Actions:**
    - The admin sent a request to remove the user.
    - The system verifies that the user is registered to the system.
    - The system verifies that the user is not an admin.
    - The user will be removed from the system.
    - The admin will receive feedback from the system.

Action	Data	Expected Result
Remove user	registered user email	user has been removed successfully from the system.
	unregistered user email	The user does not exist in the system.
	invalid email	invalid input.

Use-case 15.2 : Edit user profile

- **Actor:** Admin
- **Precondition:** User is registered to the system.
- **Parameters:** User email, new email, new name, new address.
- **Actions:**
    - The admin sends a request to edit the user profile.
    - The system verifies that the user is registered to the system.
    - The user will be removed from the system.
    - The admin will receive feedback from the system.

Action	Data	Expected Result
Edit user profile	registered user email, valid data.	user profile has been changed successfully.
	unregistered user email.	The user does not exist in the system
	invalid email/name/address	invalid input

Use-case 16 : Add award

- **Actor:** Admin
- **Precondition:** None
- **Parameters:** Minimum user rank, Coupon code, due date.
- **Actions:**
    - The admin sent a request to add an award.
    - All the registered users who have rank above parameter will get a private message from the system with the award details.

Use-case 17 : Set system configurations

- **Actor:** Admin
- **Precondition:** None
- **Parameters:** Minimum distance to alert, alert type, number of routes to offer.
- **Actions:**
    - The admin sends a request to set the system configurations.
    - The system will check validation of the parameters.

Use-case 18: Manage Admin

Use-case 18.1 : Add admin

- **Actor:** Admin
- **Precondition:** None
- **Parameters:** Email, phone.
- **Actions:**
    - The system will add the new admin details.
    - The system will save the admin who appoints the new admin.
    - The system will send an email to the new admin with an admin-app installation link & app guide.

Action	Data	Expected Result
Add admin	valid phone & valid email	admin added successfully.
	invalid phone / email.	invalid input.

Use-case 18.2 : Remove admin

- **Actor:** Admin
- **Precondition:** There are at least two admins.
- **Parameters:** Email.
- **Actions:**
    - The system will verify that the admin has the permission to remove this admin.
    - The system will record this action.
    - The system will remove the admin and not support his access to the system as admin anymore.

Action	Data	Expected Result
Remove admin	Admin email	admin removed successfully.
	non-admin email	there is no admin with this email.
	invalid email	invalid input.

System Architecture

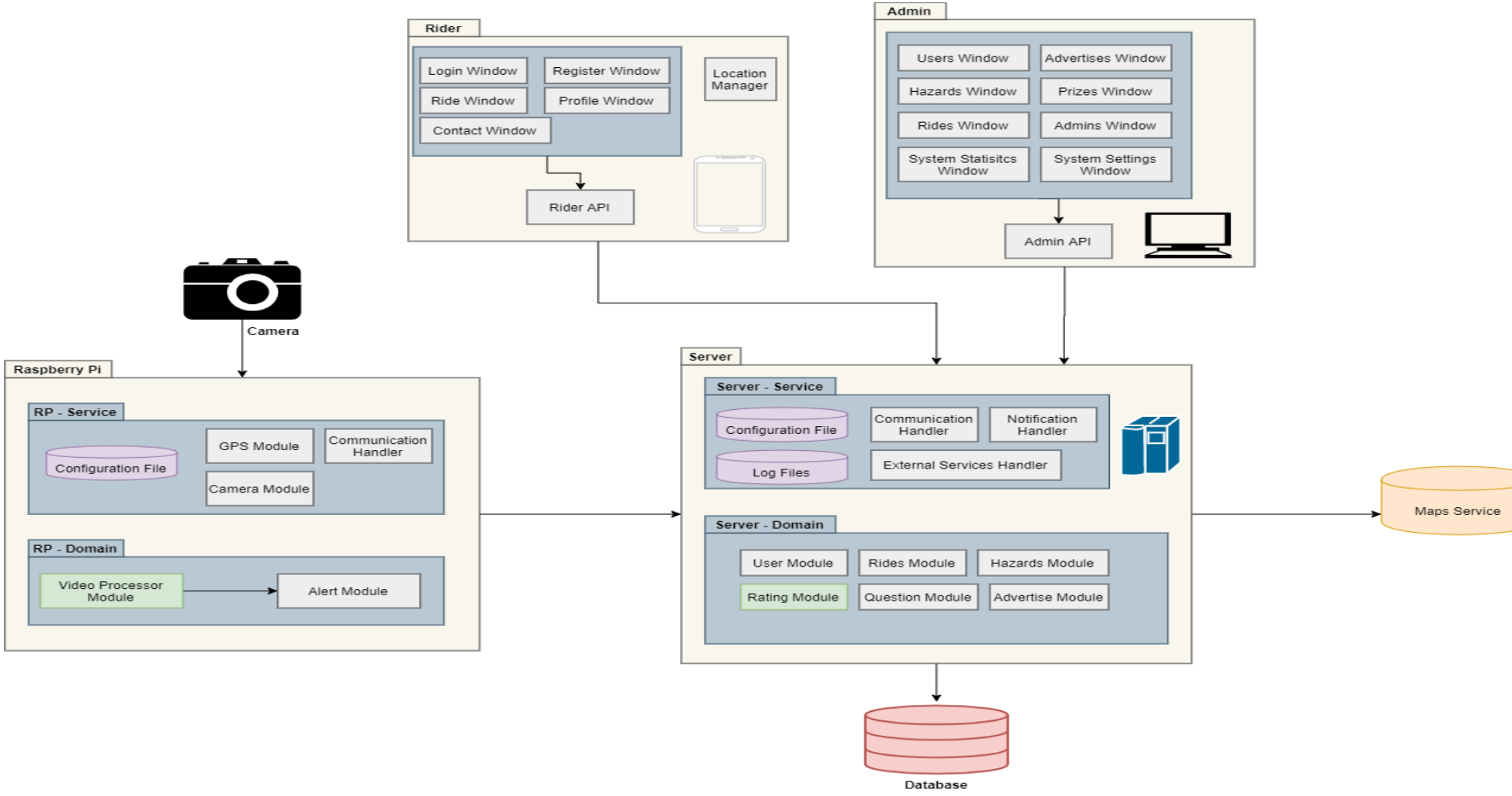


Figure 1 : System Architecture

System components:

- **Scooter** - The user's personal scooter, intended for private use only.
- **Camera** - The Raspberry Pi Camera Module 2 replaced the original Camera Module in April 2016. The v2 Camera Module has a Sony IMX219 8-megapixel sensor (compared to the 5-megapixel OmniVision OV5647 sensor of the original camera).
- **Raspberry Pi** - The component composed on the scooter, which includes Internet and GPS connections units, has the function of analyzing the frames coming from the camera, alerting if necessary, and sending data collected during the trip to a server.
- **Server** - A server that runs in a cloud environment, the unit that receives requests from the various components in the system and is responsible for saving the data and handling the various requests
- **Rider Application** - A client-side component in a mobile environment containing the sub-components: window display, communication, GPS-based location and navigation management.
- **Admin Application** - A client-side component in the Windows PC environment responsible for manipulating the information in the system, controlling the activation of various functions to change the system states.

Data Model

Description of Data Objects

Ride		
PK	ride_id	int
	rider_email	string
	date	Date
	city	String
	start_time	Time
	end_time	Time
	origin	Tuple<int, int>
	destenation	Tuple<int, int>

Ride

The object describes a ride performed by a rider registered in the system
for any such object will be save:
rider_id - the id of the rider perform the ride
date - the date the ride was made
city - the city where the ride was made
start_time - the exact time the ride started
end_time - the exact time the ride ended
origin - the exact coordinates of the origin location
destination - the exact coordinates of the destination location

Figure 2.1 : Ride Object Description

StationaryHazard		
PK	id	int
	ride_id	int
	location	Tuple<int, int>
	city	String
	type	int - (HazradType)
	size	double
	rate	int

StationaryHazard

The object describes a stationary object that appeared on a certain riding route.
The object can be a hole in the road, a light pole, an electric pole, etc
A type is basically described by the 'type' field (HazardType), in addition for any such object will be saved:
ride_id - the id of the ride in which the object was detected
location - the exact coordinates of the object
city - the city where the object is located
size - the length , width ,height of the object
rate - the rating of the hazard calculated according to the rules in appendix B

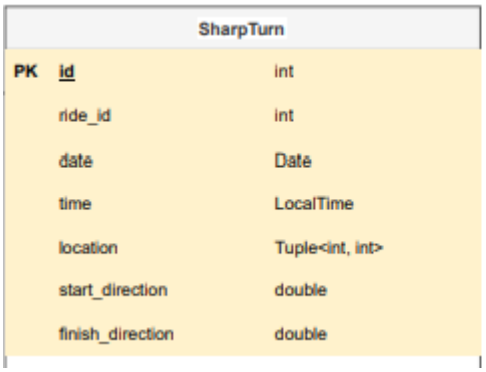
Figure 2.2 : Hazard Object Description

Brake		
PK	id	int
	ride_id	int
	date	Date
	time	LocalTime
	location	Tuple<int, int>
	start_speed	double
	finish_speed	double

Brake

The object describes a riding action in which the rider perform braking
for any such object will be save:
ride_id - the id of the ride where the action happened
date - the date the riding action was made
location - the exact coordinates in which the action was made
start_speed - the speed of the rider before the brake
finish_speed - the speed of the rider after the brake

Figure 2.3 : Brake Object Description

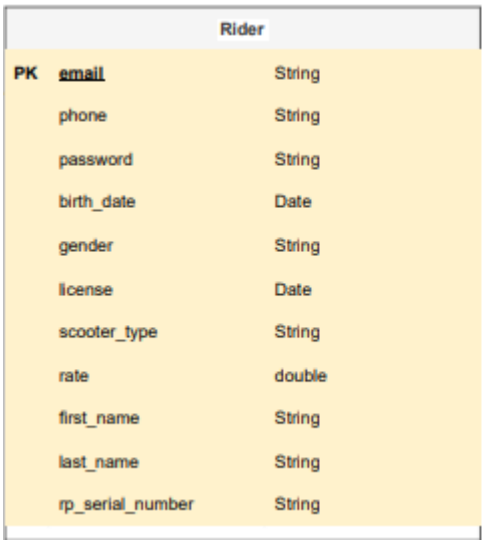


SharpTurn		
PK	id	int
	ride_id	int
	date	Date
	time	LocalTime
	location	Tuple<int, int>
	start_direction	double
	finish_direction	double

SharpTurn

The object describes a riding action in which the rider perform sharp turn
for any such object will be save:
ride_id - the id of the ride where the action happened
date - the date the riding action was made
location - the exact coordinates in which the action was made
start_direction - the direction of the ride before the turn
finish_direction - the direction of the ride after the turn

Figure 2.4 : Sharp-Turn Object Description

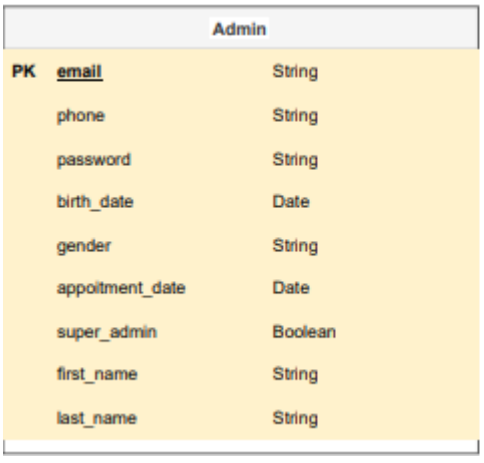


Rider		
PK	email	String
	phone	String
	password	String
	birth_date	Date
	gender	String
	license	Date
	scooter_type	String
	rate	double
	first_name	String
	last_name	String
	rp_serial_number	String

Rider

The object describes a registered rider
for any such object will be save:
email , name, phone number, hash value of the password, birthdate, gender
license - the date the license was issued
scooter type - the type of rider's scooter
rate - the rating of the rider calculated by the rules in appendix C

Figure 2.5 : Rider Object Description



Admin		
PK	email	String
	phone	String
	password	String
	birth_date	Date
	gender	String
	appoitment_date	Date
	super_admin	Boolean
	first_name	String
	last_name	String

Admin

The object describes an Admin of the system
for any such object will be save:
email , name, phone number, hash value of the password, birthdate, gender
appoitment_date - the date the admin was appointed
location - the exact coordinates in which the action was made
super_admin - wether the admin is super admin or not

Figure 2.6 : Admin Object Description

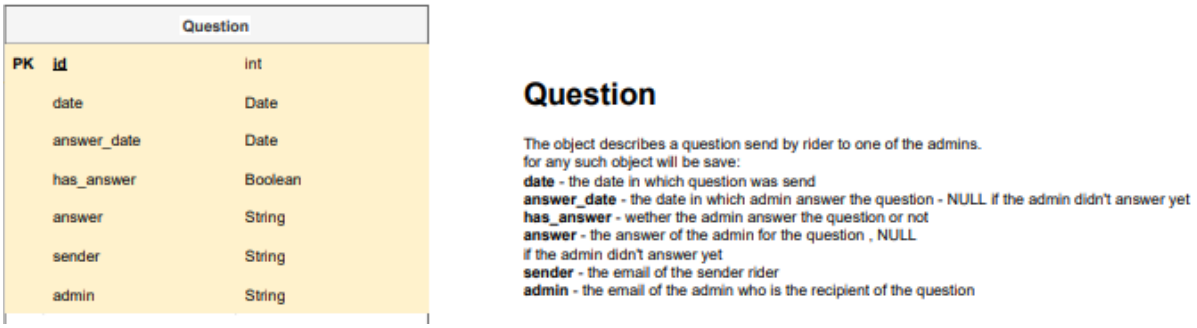


Figure 2.7 : Question Object Description

Figure 2.8 : Advertise Object Description

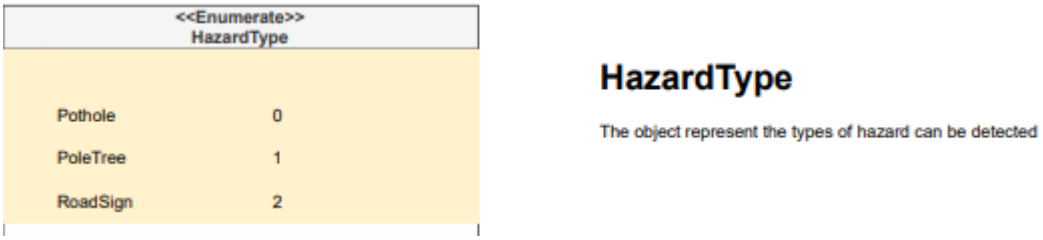


Figure 2.9 : Hazard Type Object Description

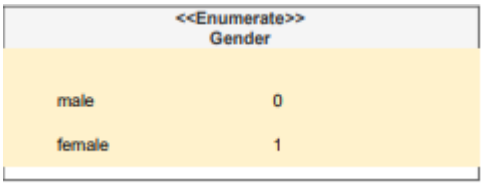


Figure 2.10 : Gender Object Description

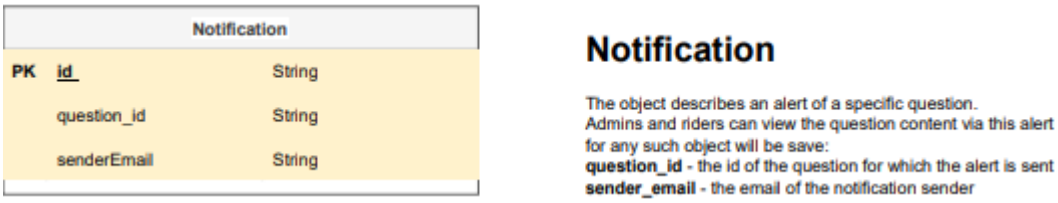


Figure 2.11 : Notification Object Description

Data Objects Relationships

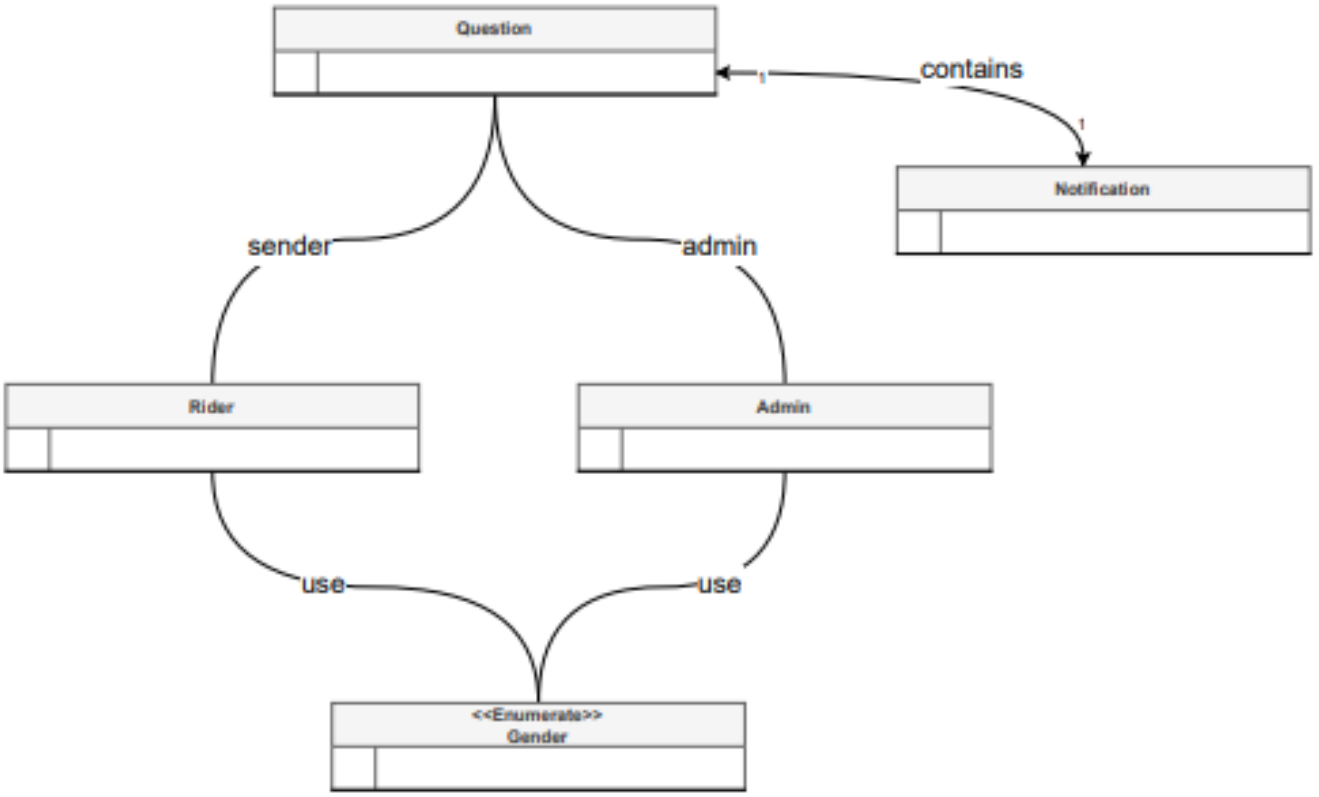


Figure 2.12 : Data object Relationships -Part A

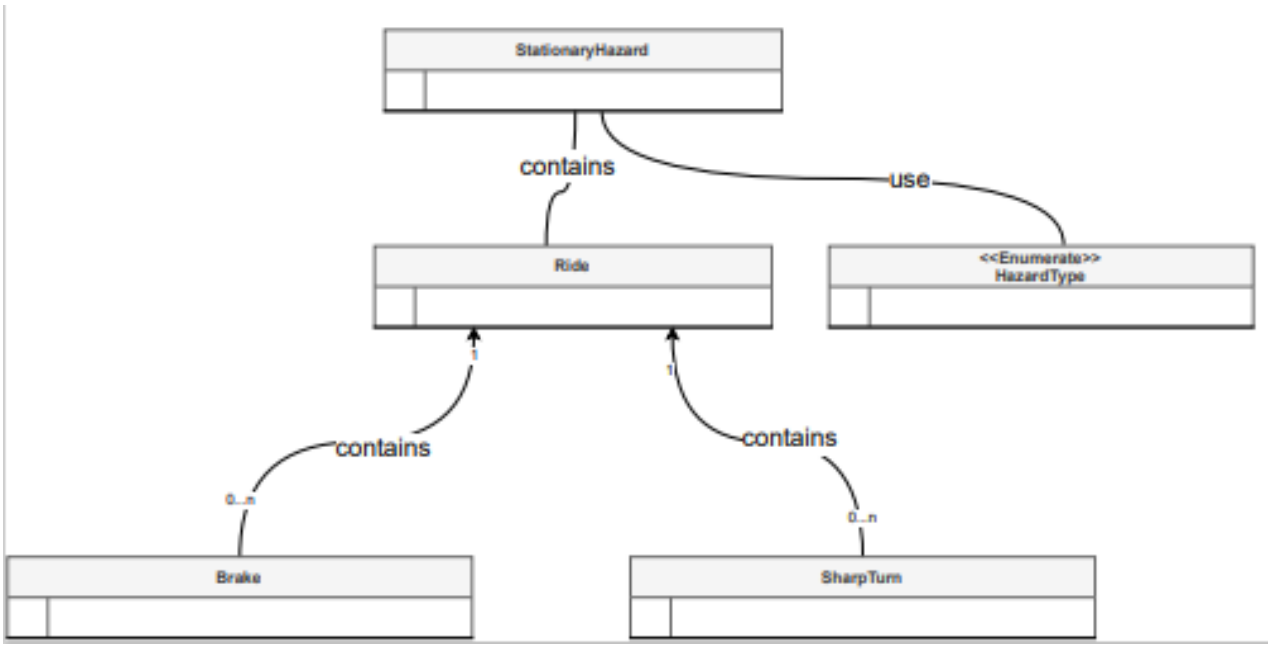


Figure 2.13 : Data object Relationships -Part B

ERD

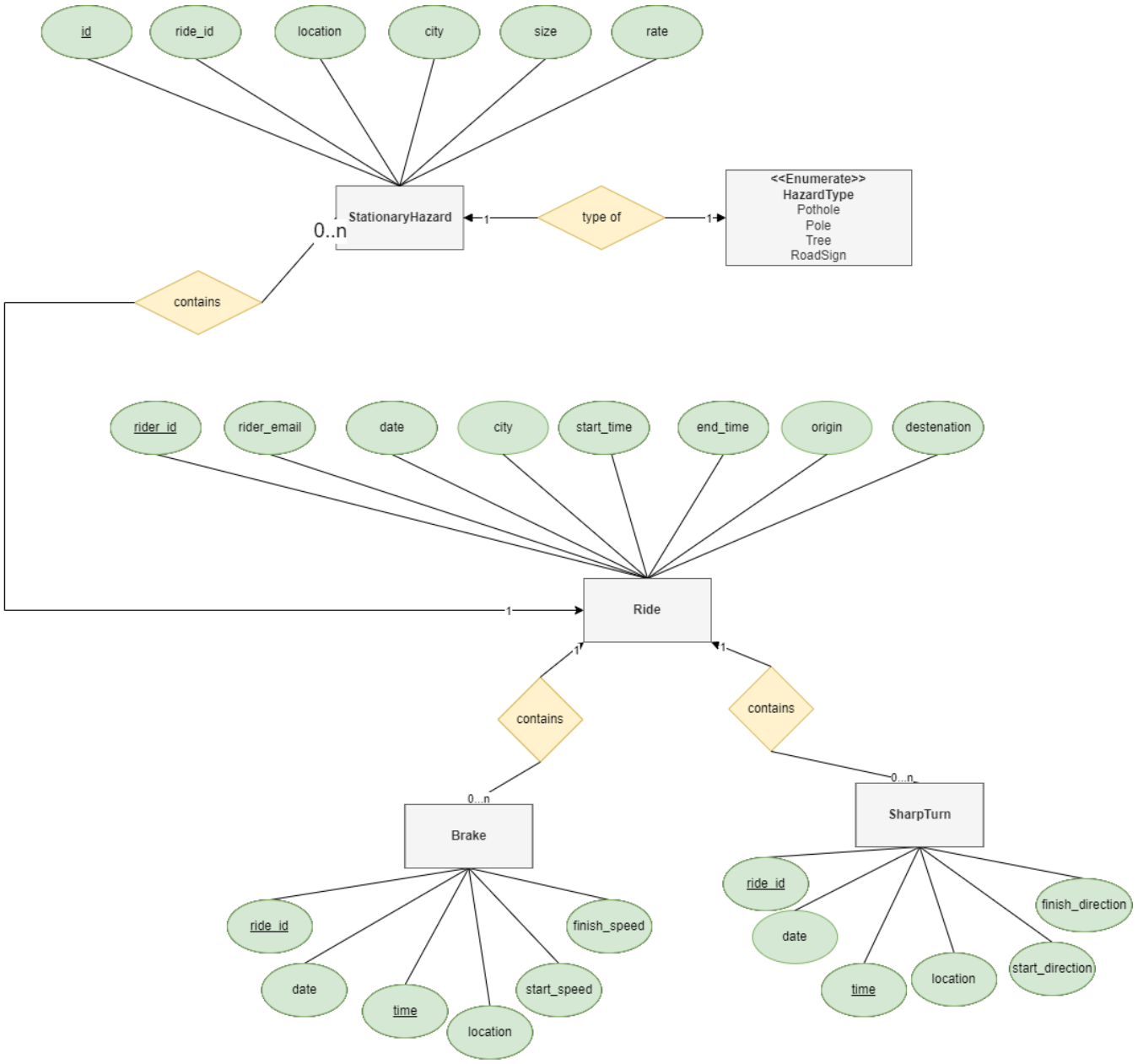
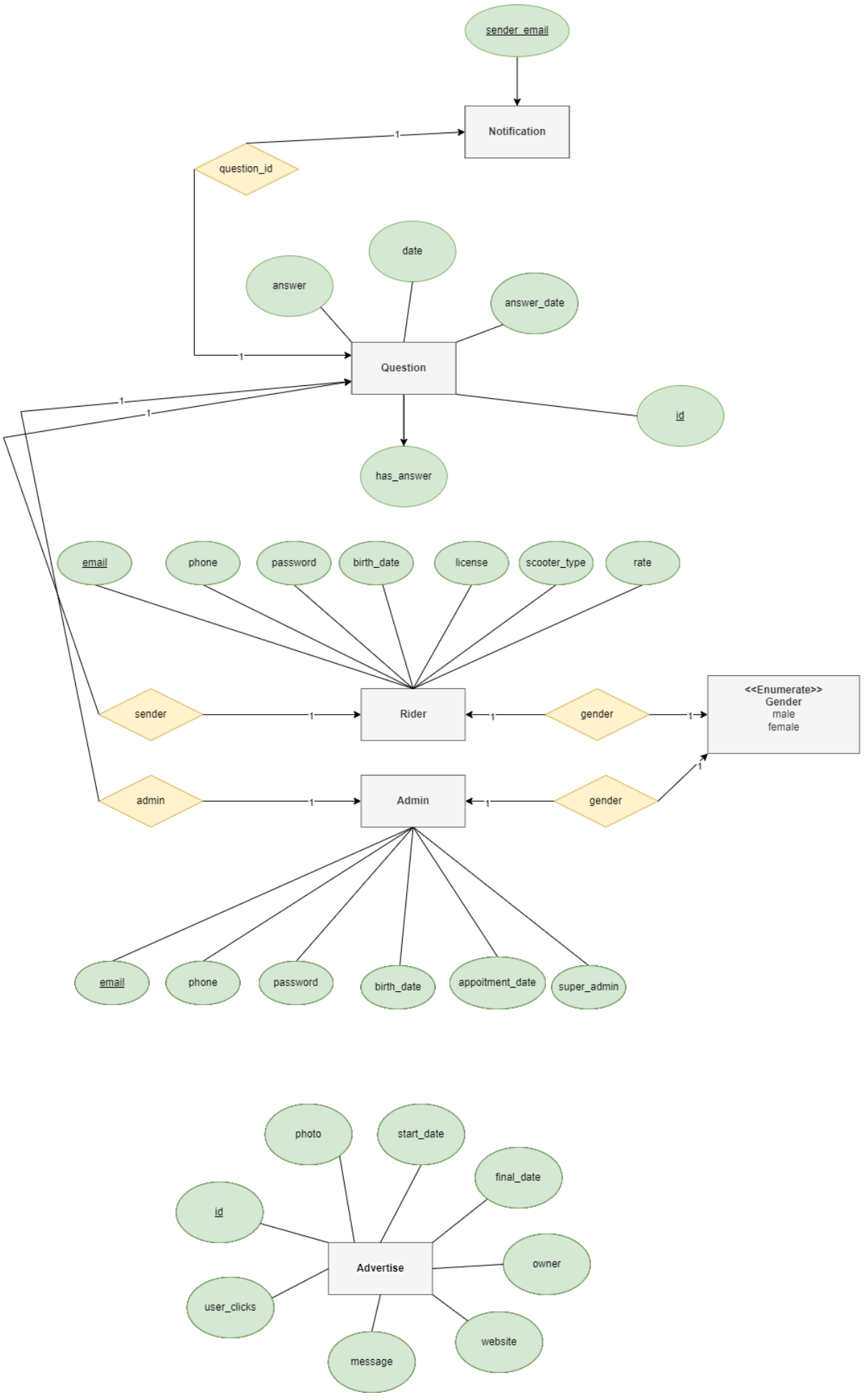


Figure 3.1 : ERD part 1


sender_email
Notification
1
question_id
date
answer
answer_date
Question
1
1
id
has_answer
email
phone
password
birth_date
license
scooter_type
rate
sender
1
Rider
1
gender
1
<<Enumerate>>
Gender
male
female
admin
1
Admin
1
gender
1
email
phone
password
birth_date
appoitment_date
super_admin
photo
start_date
final_date
id
Advertise
owner
user_clicks
website
message

Behavioral Analysis

Events

- **Server Initialization Event** - Upon system initialization it will load information from a configuration file, and will fetch information from the database if it exists.
  If there is no information in the database which means it's the first initialization of the system then the system will simply create all classes and appoint default admin.

- **Raspberry Pi Initialization Event** - Upon raspberry pi initialization it will load information from a configuration file, start the camera and start recording video.

- **Server Shutdown Event** - Upon receiving a shutdown event the Server will finish any computations that were in the process of being done, save all information to the database, write information regarding the shutdown to the logger and shutdown gracefully.

- **Finish Ride Event** - Upon finishing a ride the raspberry pi will gather all of the information he gathered during the ride and will send the information to the server which in turn will update the system according to information.
  The server will save the information about new hazards that were discovered during the ride, update rider rating based on the rider behavior during the ride and will store information about the ride in the system such as: date, source, destination.

- **Hazard Detected Event** - Upon identification of a hazard while driving the raspberry pi will store information regarding the hazard such as: location, type, time of detection.
  The raspberry pi will accumulate the information during the ride and will send it to the server upon finishing the ride (Finish Ride Event)

States

- **Server up** - A state where all system components are supposed to work, when the server provides its services and responds to every request, the RP works as an independent unit and at the end of the calculation (trip) sends the data to the server.
  For this mode, the developers should run the server application on the cloud and provide the host & port data.

- **Server up with no data** - A state who describes the first use of the system, when no data from the database should be load,
  the system will be in this mode in two cases:
  when the server is running for the first time
  after a developer called the 'reset' method.

- **Server down** - A state that each RP unit should work independently, when the ride is finished, the RP will save the data about the ride and wait to send it until a connection is established.
  In this state, the rider application should display a message explaining that the service is currently unavailable while the admin application should provide an explanation of the reason for the service interruption.
  For this mode, the developer should apply the 'shut down' method.

- **RP Connect -** A state when the RP is successfully connected to the server and transmit data after finish rides.

- **RP Disconnect** - A State who consist of the two following sub-states:
  1. The RP has been initialized by the company, although the user did not complete the registration process, that is, the user has received his personal serial number from the company but has not yet completed the registration process in the rider application.
  2. The RP fails to establish a connection to the server.

  In this situation, the RP system should work independently but the server should warn of data loss in this case.

- **RP Malfunction** - A situation where the RP is malfunctioning, in this situation the care of a technical person capable of performing tests on all the different systems of the RP is required.

Changing system states methods:

- **Reset:**
  precondition : ‘Server up’ mode
  postcondition : ‘Server with no data’ mode

- **Shut down:**
  precondition : ‘Server up’ mode
  postcondition : ‘Server with no data’ mode

- **Register:**
  precondition : ‘RP Disconnect’ mode
  postcondition : ‘RP Connect’ mode

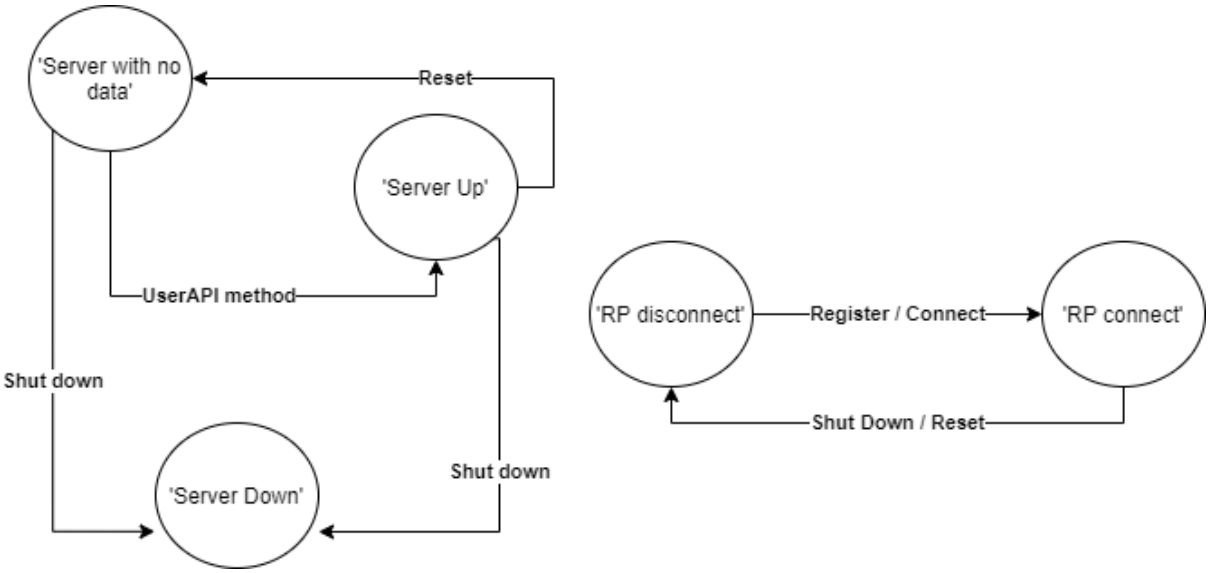


Figure 5 : States Diagram

Object-Oriented Analysis

Class Description

Packages

Internal Packages

Server-side:
The server side consists of several layers that are designed vertically in our system,
the layers that exist on the server are:

- **API Interfaces** - contains interfaces for requests from the various systems: admin, rider and Raspberry Pi.
- **Service layer** - the interface layer between the external components and the business layer, including the following packages:
    - Communication package - the package that is responsible for managing the communication with the various components of the system, includes implementation of the API using the Rest API.
    - External Services package - have the responsibility for connect and send requests to the Google maps api, including handling responses range answers, implemented with proxy and adapter.
    - Initialization package - includes the configuration files and the "server initialization" class that includes the following key methods:
        - Set static vars - initialize static variables from a configuration file.
        - Connect external services - a method responsible for creating a connection to the external services (Google Maps) using connection details found in a configuration file.
        - Create rp config file - a method that creates a configuration file that the server will send to the various components for a default initialization of settings.
        - Connect database - a method that is responsible for creating a connection to the database using connection information found in a configuration file.

- **Business layer** - holds most of the logic in the system, includes the specific algorithms for rating users and dangers.
  including the following packages:
    - Advertise Module - A module who manages the data about the advertisements in the system.
    - Hazards Module The module that handles requests to add, edit and delete hazards, in addition, contains enum Hazard Type who sets the domain of hazards the system supports.
    - Questions Module -The module who handles questions from the users to admins, and notify when an answer arrives from an admin.

- Rating Module - the core of the server-side, implements the ideas about rating users and hazards in our system.
  Main classes:
  - Hazard Rate Calculator - The role of this class is to rate the various dangers that may be found in a rider's path.
    According to the danger score, which is based on various parameters such as the width of the danger in relation to the width of the road,
    The system will be able to estimate the safety level of different roads.
  - User Rate Calculator - The role of this class is to rate the level of safety of a rider based on the trips he has made.
    According to the rating of the riders, it will be possible for the system administrators to receive a business situation assessment regarding transactions with the various riders.
  - Routes Retriever- The class who should get the data from google maps and sort the routes according to the hazards rates inside each route.
- Users Module - The unit which manages both users & admins.

- **Data access layer** - a layer that is the separation between the database and the system, manages the execution of queries to the database.

- The **Utils package** is used by the different layers in the code and its role is to provide static function services.
- A **logger package** that is divided into three loggers that write to text files:
  - Error logger - responsible for documenting all errors that the system detected and handled
  - System logger - responsible for documenting system initialization, contacting external services, creating a connection with the database, initializing configuration files
  - Server logger - records all the requests the server received.

Raspberry Pi:
Raspberry Pi contains different layers:

- **Service layer** - its role is to initialize the system settings according to configuration files and create a connection with the server, through which the information will be transferred. Contains the two central modules:
  1. Communication package
  2. Configuration package
- **Domain layer** - implements the main logic of the component and its role is to receive input from the camera, process the image and alert if necessary. Contains the following packages:
  1. Alerts module - responsible for outputting input to the user by various means
  2. Camera module - responsible for receiving input from the camera
  3. GPS module - responsible for receiving the location of the component

4. Image analysis module - the main logic of the system, its role is to receive an image and return an answer - "is there a danger in the image".

   This module includes the following main classes:

   - Hazard Detector - The role of this class is to process frames and decide if the frame contains hazard, in addition, the class should classify the hazard from the range of possibilities that the system supports.
   - Road Detector - This class should classify the route based on the first frame that arrived from the camera.

External Packages

OpenCV provides a real–time optimized Computer Vision library, tools, and hardware. It also supports model execution for Machine Learning.

React Native combines the best parts of native development with React, a best-in-class JavaScript library for building user interfaces.

Java Spring Framework provides a comprehensive programming and configuration model for modern Java-based applications - on any kind of deployment platform.

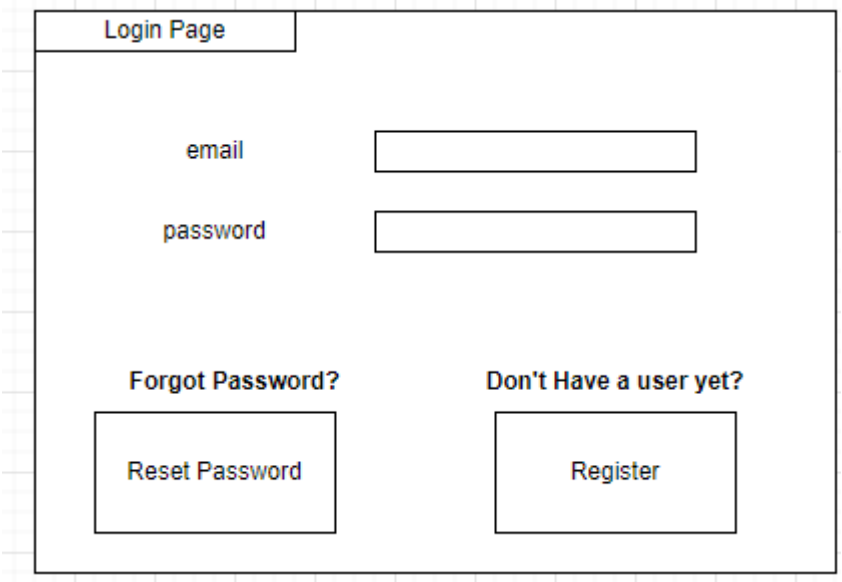
Gpsd is a service daemon that monitors one or more GPSes or AIS receivers attached to a host computer through serial or USB ports, making all data on the location/course/velocity of the sensors available to be queried on TCP port 2947 of the host computer.
https://maker.pro/raspberry-pi/tutorial/how-to-use-a-gps-receiver-with-raspberry-pi-4

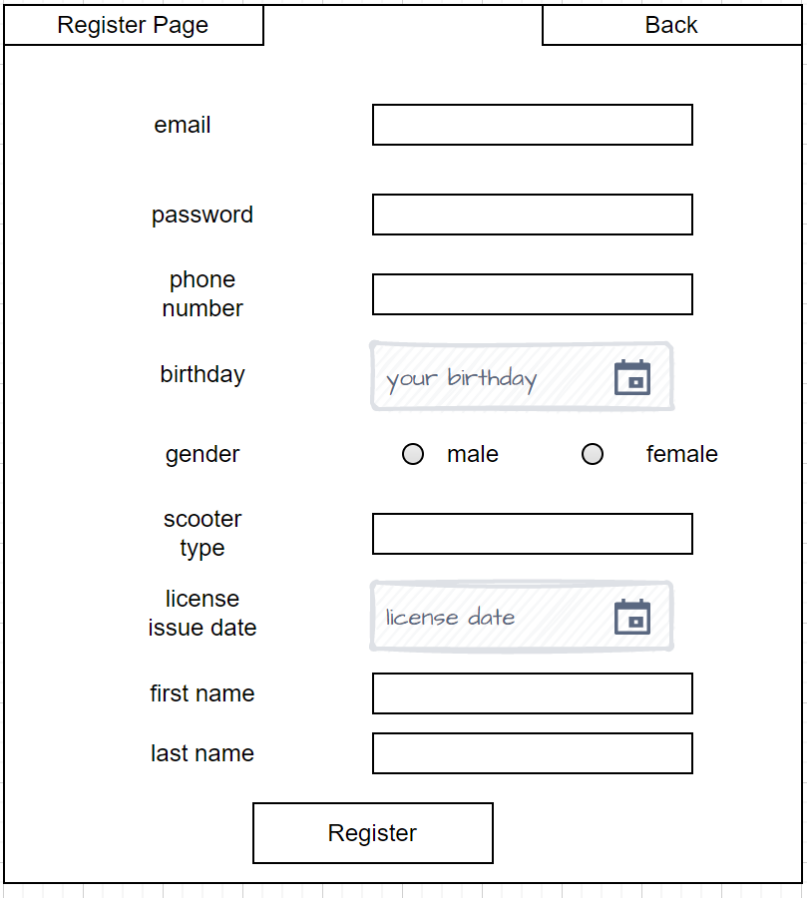
Google-maps-services-java This library brings the google maps api web services to our server-side Java application.
https://github.com/googlemaps/google-maps-services-java

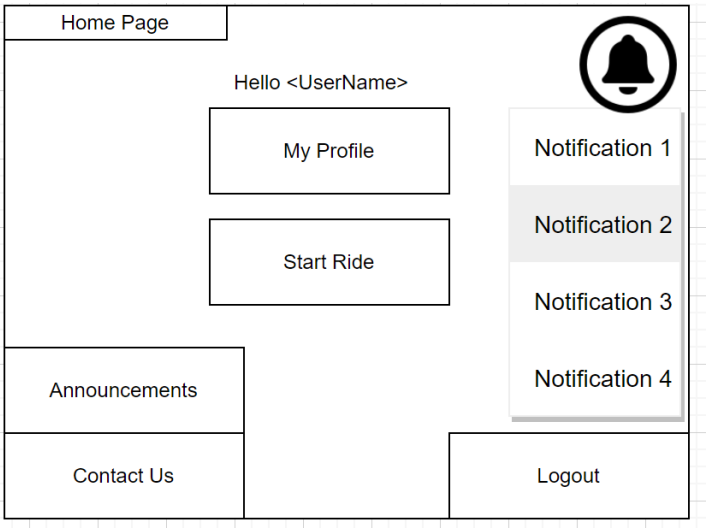
Hibernate is an object–relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate handles object–relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.

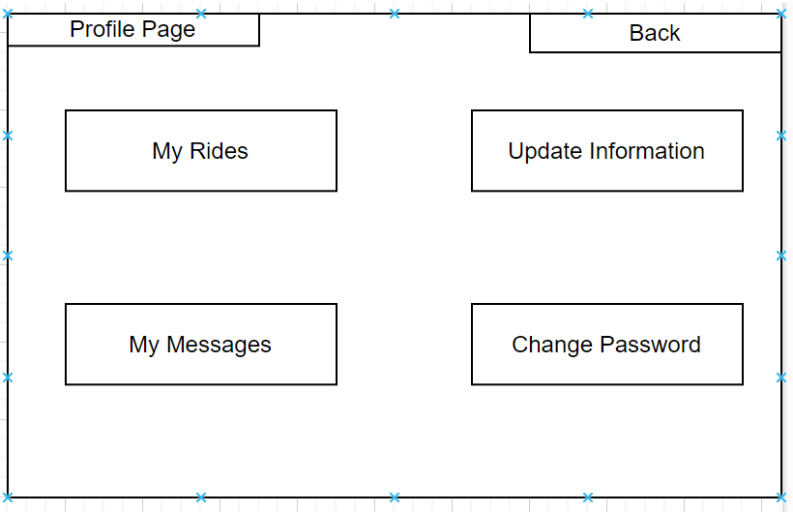
User Interfaces

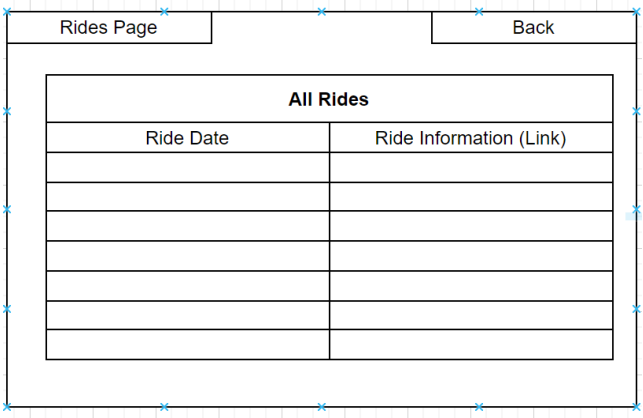
User Interfaces Draft

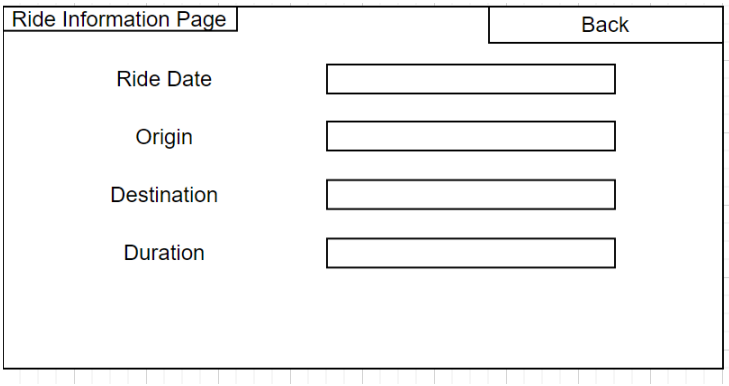


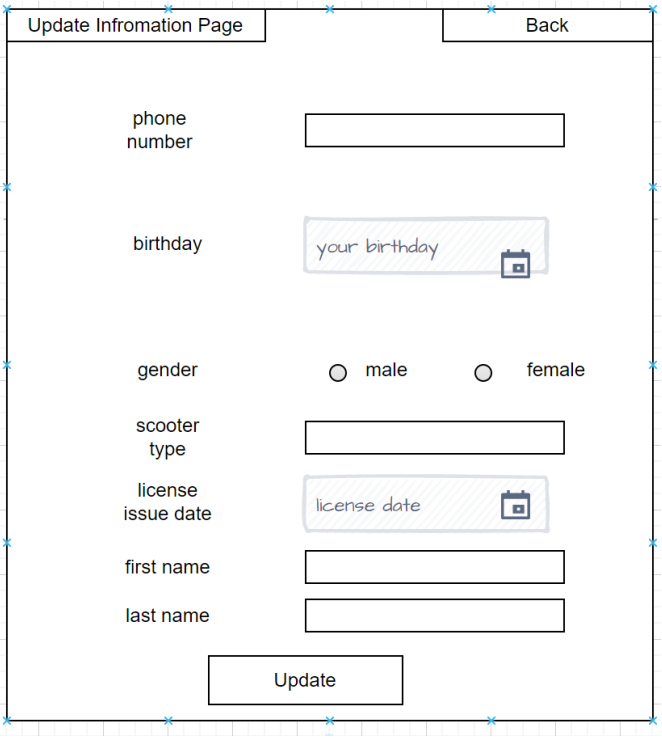


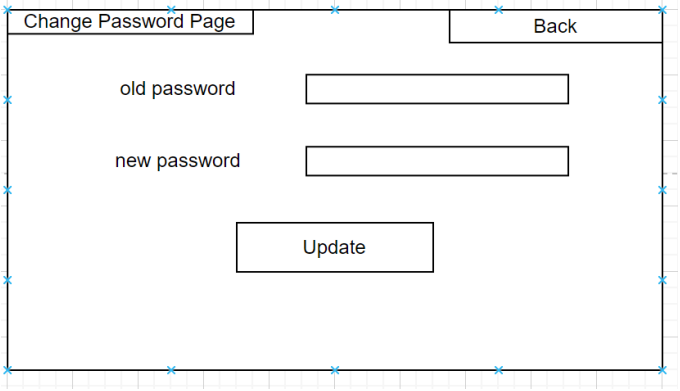

Home Page
Hello <UserName>
My Profile
Start Ride
Announcements
Contact Us
Logout
Notification 1
Notification 2
Notification 3
Notification 4

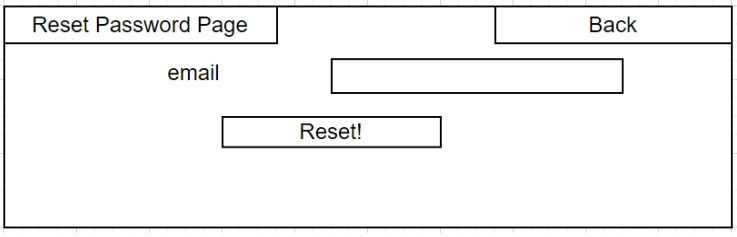

Profile Page
Back
My Rides
Update Information
My Messages
Change Password

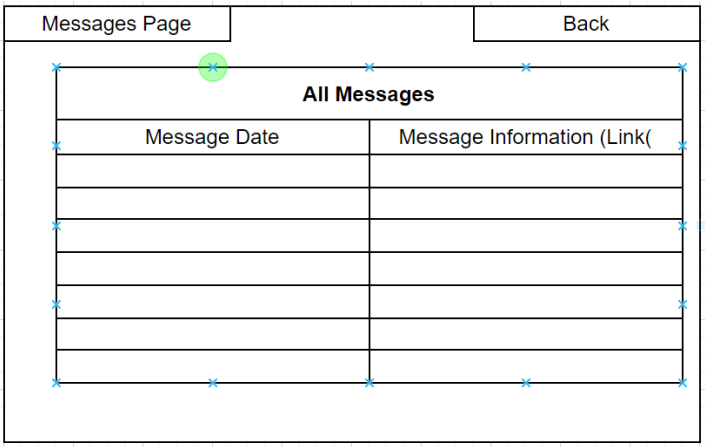

Rides Page
Back
All Rides
Ride Date
Ride Information (Link)

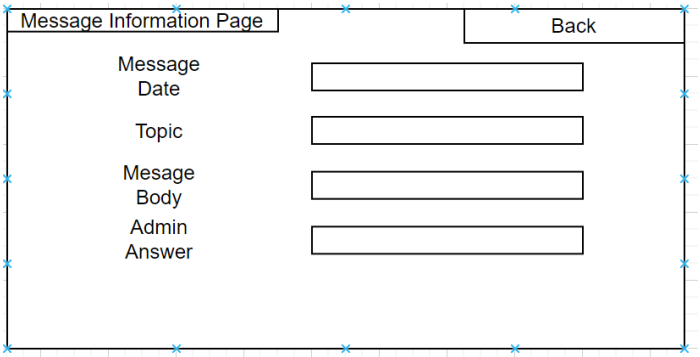

Ride Information Page
Back
Ride Date
Origin
Destination
Duration

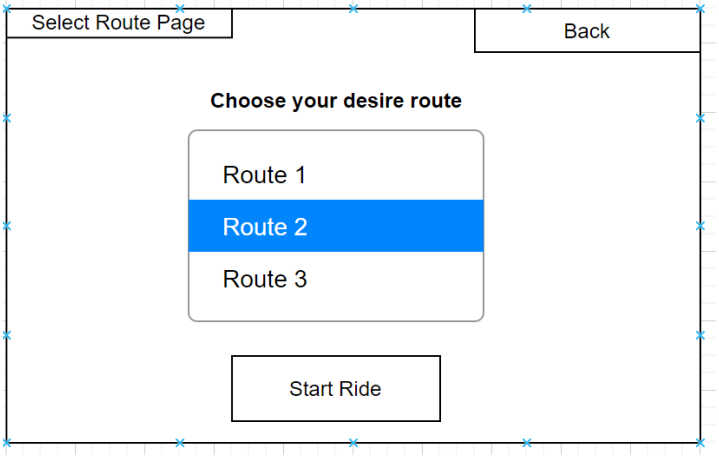

Update Infromation Page
Back
phone number
birthday
your birthday
gender
male
female
scooter type
license issue date
license date
first name
last name
Update

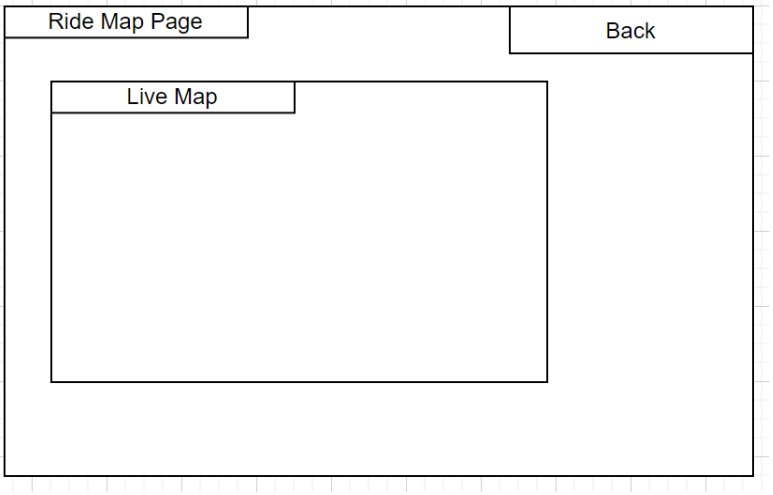

Change Password Page
Back
old password
new password
Update

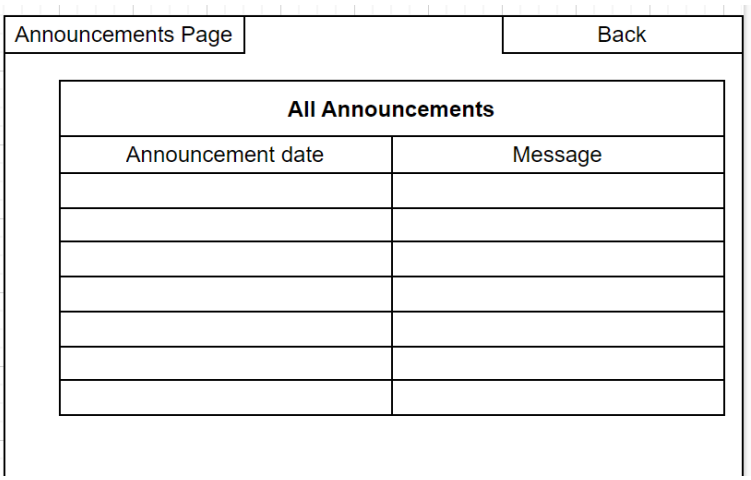

Reset Password Page
Back
email
Reset!


Messages Page
Back
All Messages
Message Date
Message Information (Link(


Message Information Page
Back
Message Date
Topic
Mesage Body
Admin Answer


Select Route Page
Back
Choose your desire route
Route 1
Route 2
Route 3
Start Ride


Ride Map Page
Back
Live Map


Announcements Page
Back
All Announcements
Announcement date
Message

Admin Interfaces Draft

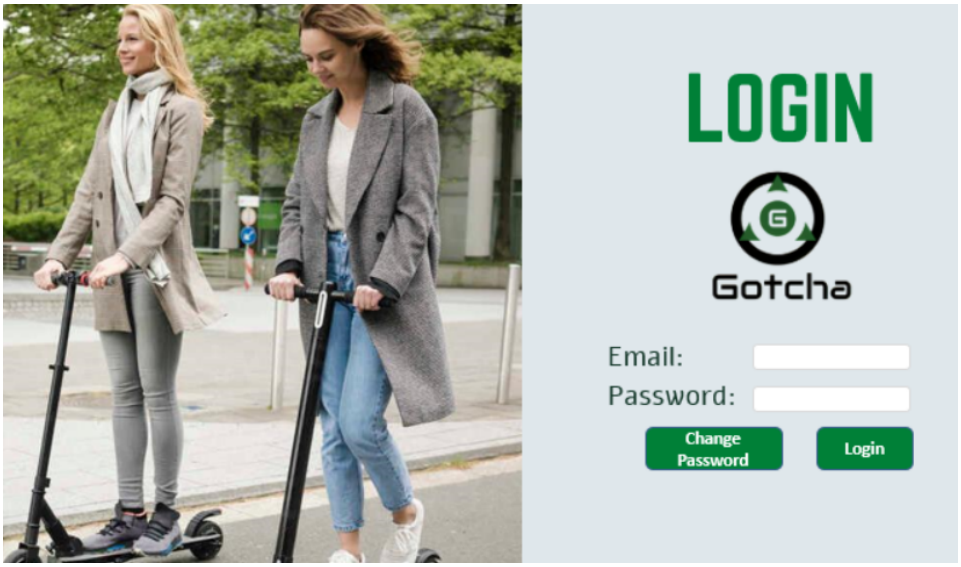


Figure 8.1 : Login window

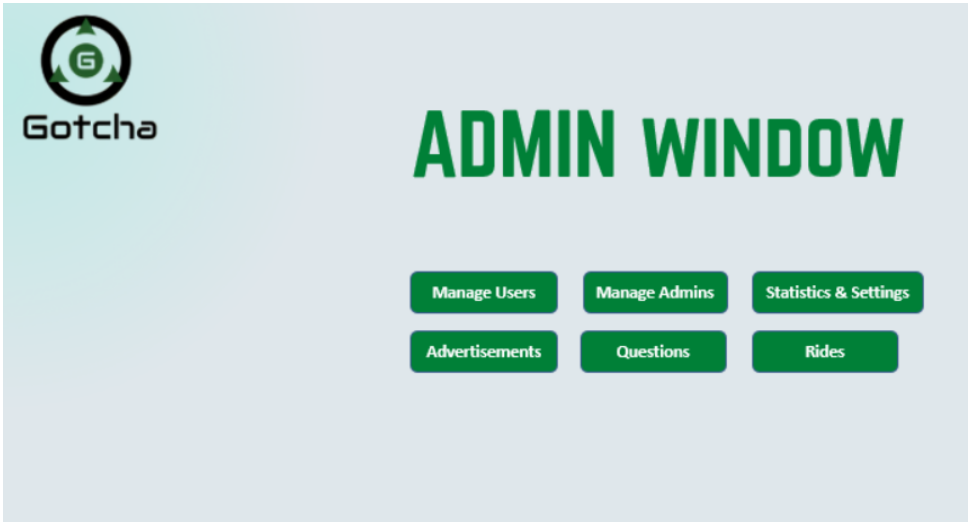


Figure 8.2 : Admin main window

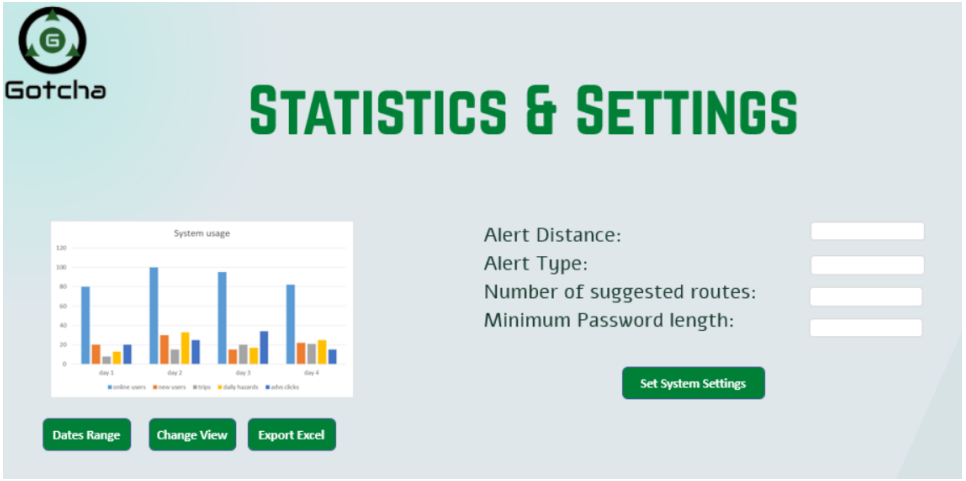


Figure 8.3 : Statistics & Settings window

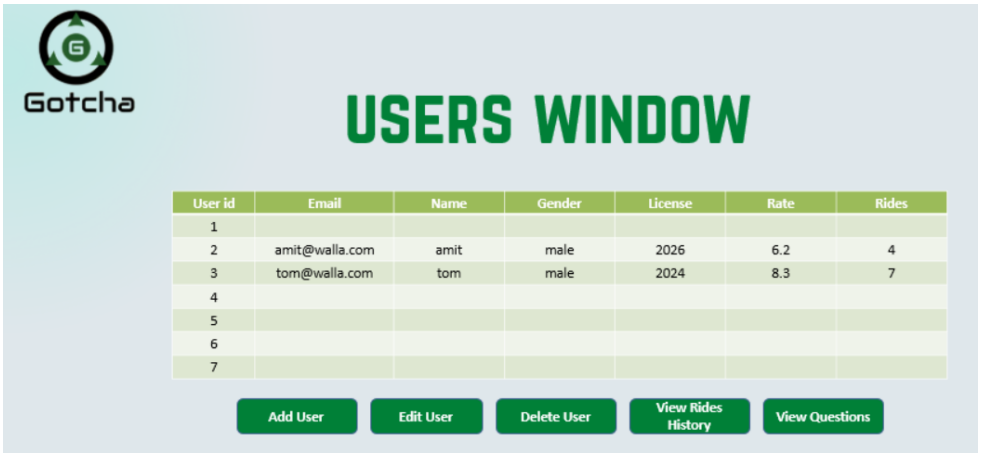


Figure 8.4 : Users window

Figure 8.5 : Rides window

Testing

1. Unit test:

Our unit tests plan for the following classes & modules according to preconditions and postconditions without considering the other system components.

Server:

- Advertise module - advertise controller: add, remove & edit advertisements.
- Hazards module - hazards controller: add, remove & edit hazards.
- Questions module - question controller: add, remove, edit & answer questions.
- Users module - users controller: add admin, change password, login, logout & register.
- Rating module -rate controller: rate rider, rate hazard.
- Utils - check all methods.
- Logger - write to loggers.
- Password manager - encrypt and decrypt passwords.

Raspberry Pi:

- Image processor module - detect all hazards types in a frame.
- Alert module - verify verbal and visual outputs from RP (manual).
- Camera Module - we will be testing the “get next frame” method.
- GPS module - we use Gpsd module, therefore, we will based on the package tests.

2. Acceptance tests:

The acceptance testing plan for the project will be performed per the usage scenarios using the data tables containing happy, sad and bad scenarios.
In accordance with the actors that activate the action, we will separate the tests into tests that are activated by each actor when the admin application, the rider application and the Raspberry Pi will activate the usage scenarios.

As part of our testing program, we will abstractly describe the Facade class on the server that implements the "complete functions" (that is, functions that are activated from another component and activate a set of sub-functions), as an endpoint that initiates the use case scenario.

For this tests, we intended to implement the following design patterns combination for separate system tests & implementation:

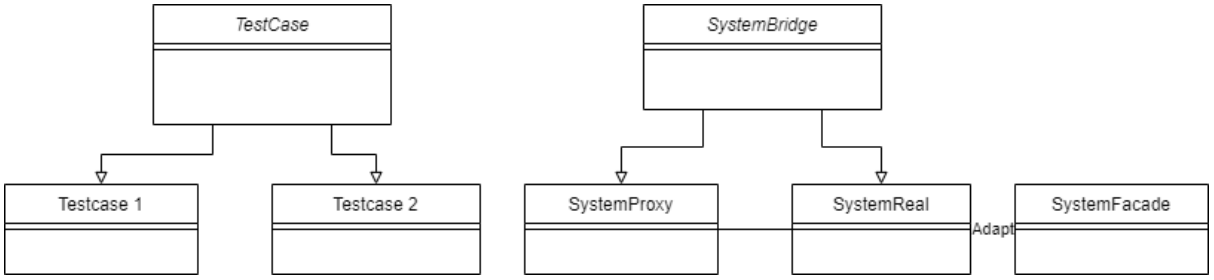


Figure 9.1 : Acceptance tests design

3. Integration tests:

Server - we will use bottom up methodology to test our integration,
We will use Mocks for testing each unit, and incrementally connect the separate units until we get complete integration of the server.

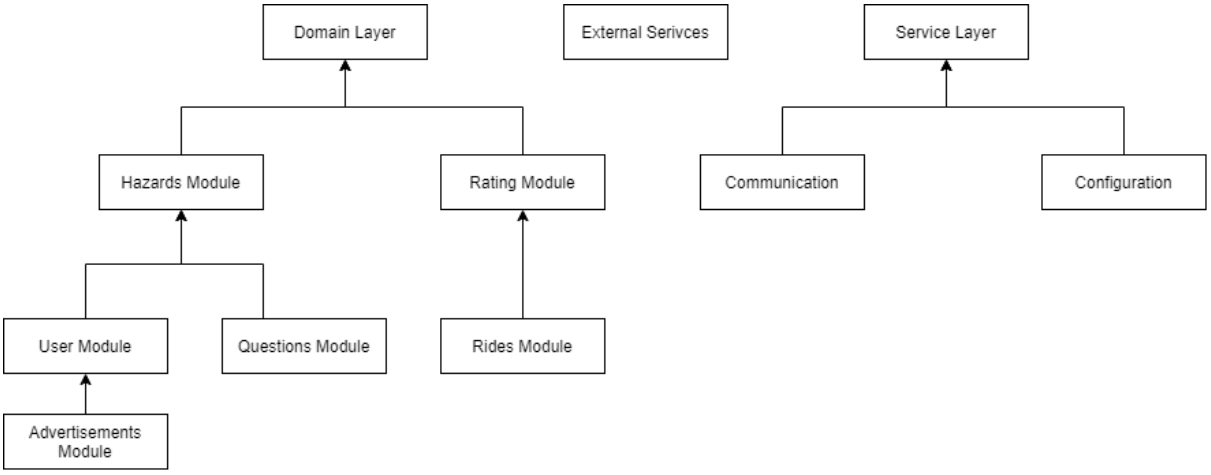


Figure 9.2 - Server bottom up integration tests design

Raspberry Pi - we will use bottom up methodology to test our RP integration, our main challenge is to test Camera & Alert modules automatically.
In addition, our video process module will test with our tagged images hoard.

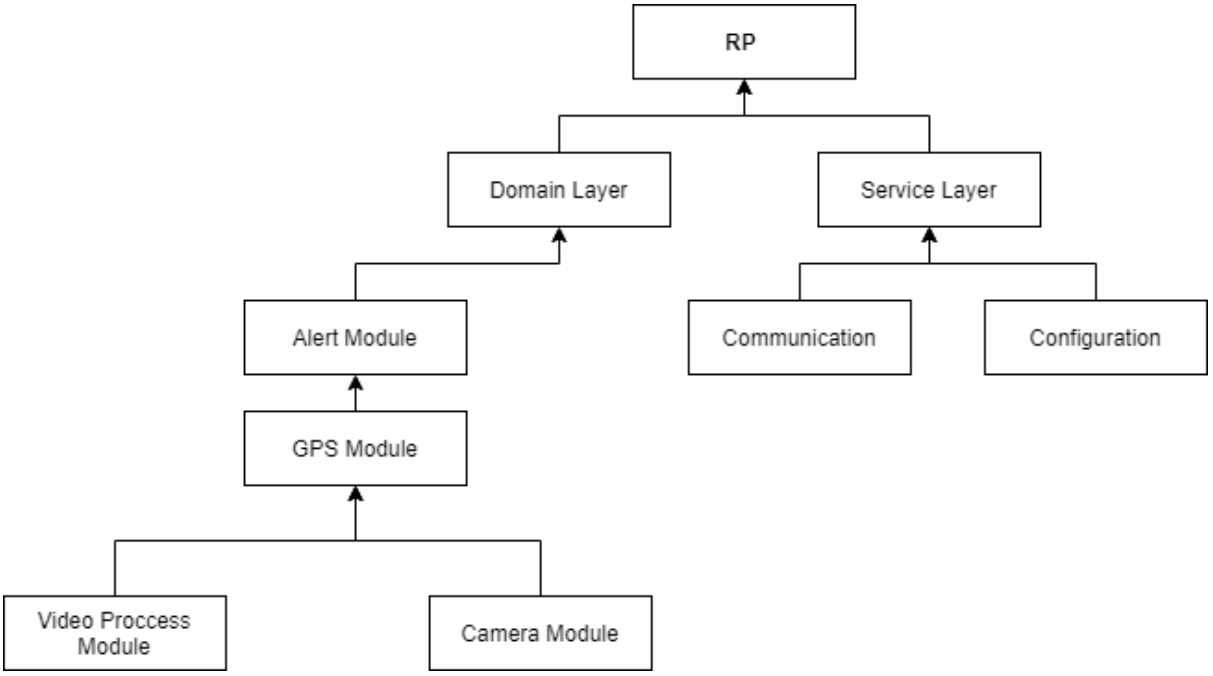


Figure 9.3 - RP bottom up integration tests design

4. System tests:

Our system tests will focus mostly on raspberry pi before we release it, and we will test the use cases of which rider is the actor.

In these tests, we will use all the real system components and units(except external services) and test all the Raspberry pi functionality.

The company should give us a road with all hazards types and we expected to get an alert for each one of them.

In this test, we use all the different options for Raspberry pi's configuration.

In addition, our team will check the admin & rider applications with different data for each use case.

5. Load tests:

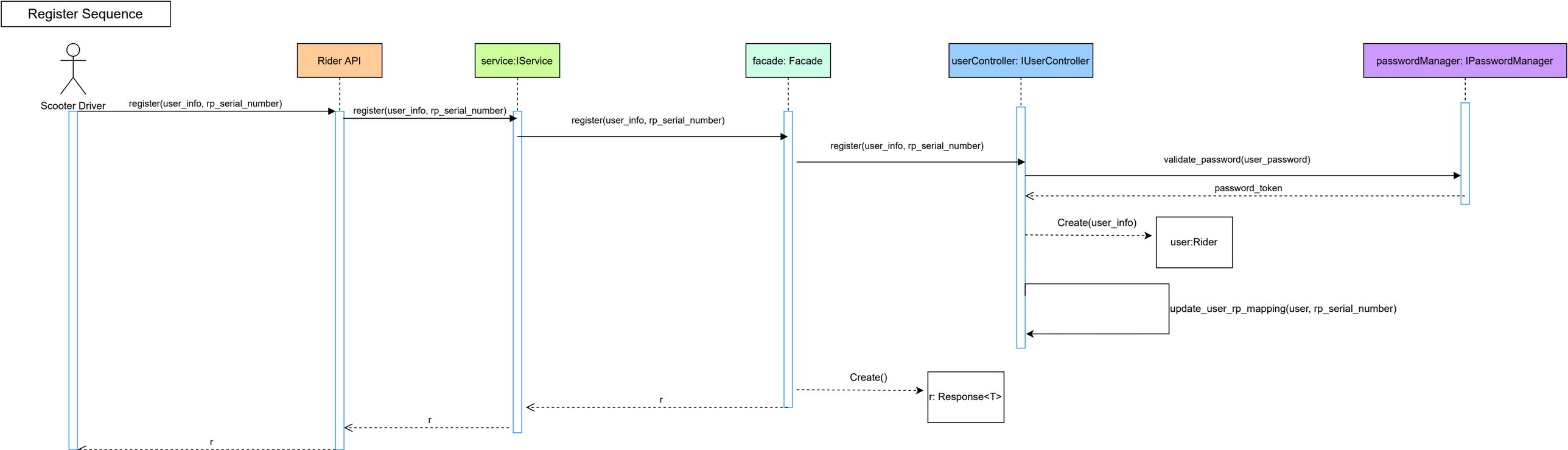
According to 37 + 38 requirements, we will implement load tests in the server which includes handling 50,000 register requests(not at the same time), and 10,000 login requests at the same time - which means 100 login requests per minute during 10 minutes.

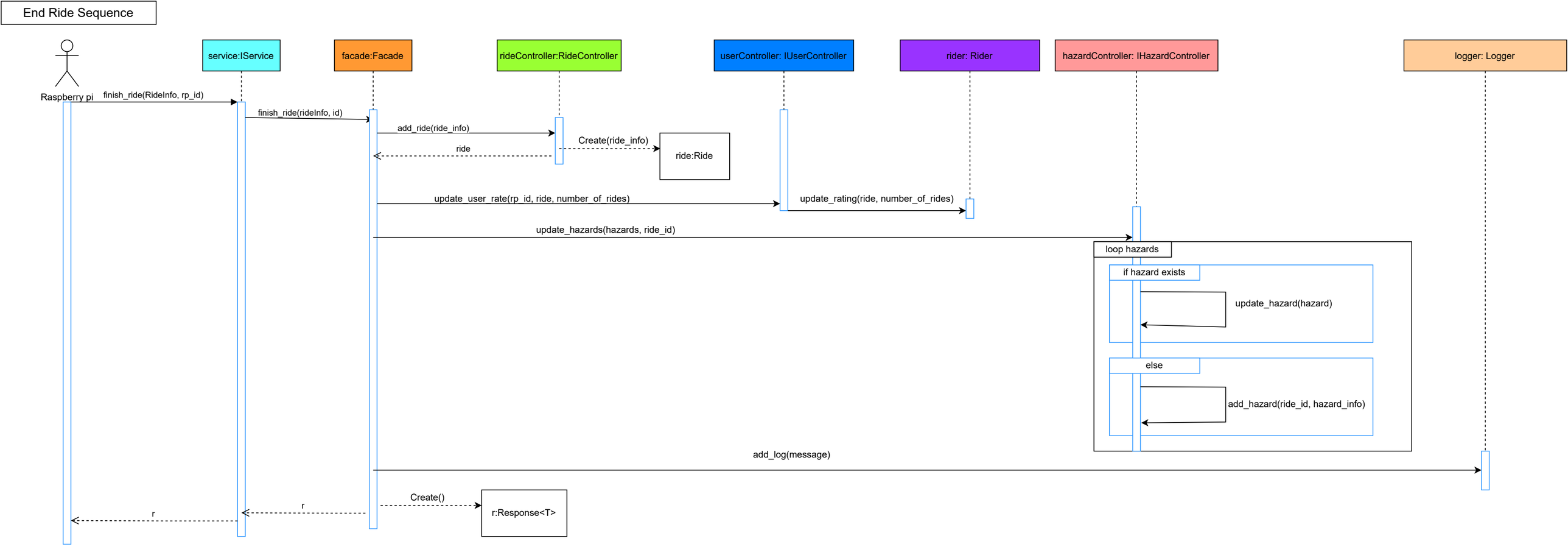
As of this writing, we plan to automatically examine the log files by searching and counting error messages, but other technological options will be examined.

number of requirement	name / description	type	input	expected output	status	comments
3	Register					
5	Login					
6	Logout					
7	send question					
8	get routes					
9	ride instructions					
10	detect hazards and alert					

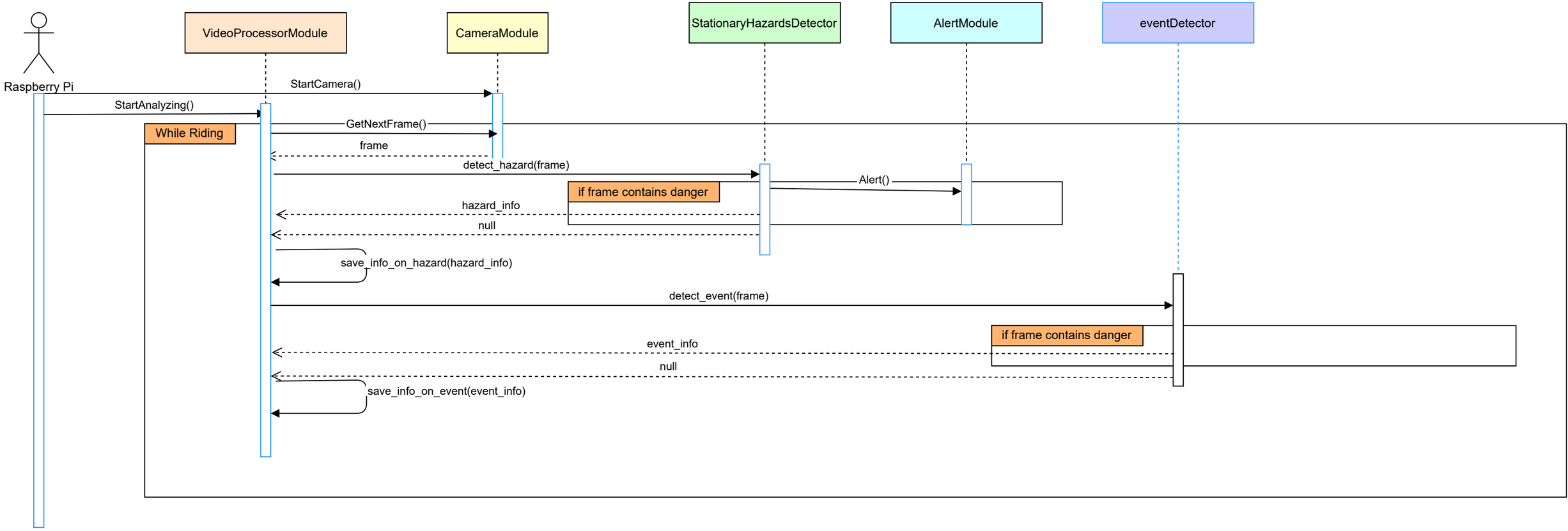
Non Functional Requirements Test plan:

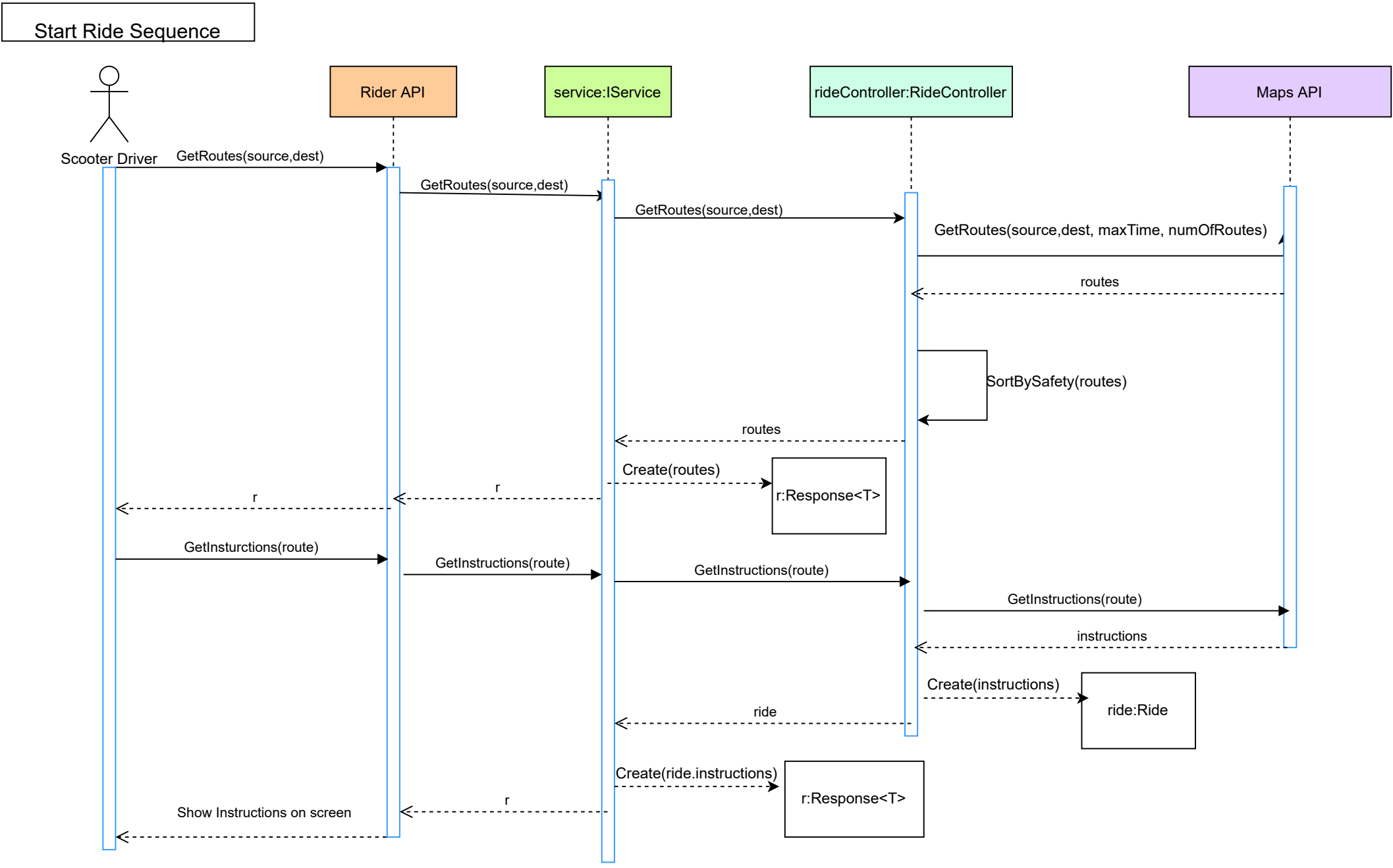
Requirement	How we are going to test
27	We will activate the object detection system 4 times on a video of route containing a dangerous object (which has been tested as such). For each session, we will change the configuration settings so that the system creates a different type of alert. The expected result is that the type of alert that the system generates corresponds to the type of alert that it was supposed to generate
28	


Register Sequence
Scooter Driver
Rider API
service:IService
facade: Facade
userController: IUserController
passwordManager: IPasswordManager
register(user_info, rp_serial_number)
register(user_info, rp_serial_number)
register(user_info, rp_serial_number)
register(user_info, rp_serial_number)
validate_password(user_password)
password_token
Create(user_info)
user:Rider
update_user_rp_mapping(user, rp_serial_number)
Create()
r: Response<T>
r
r
r

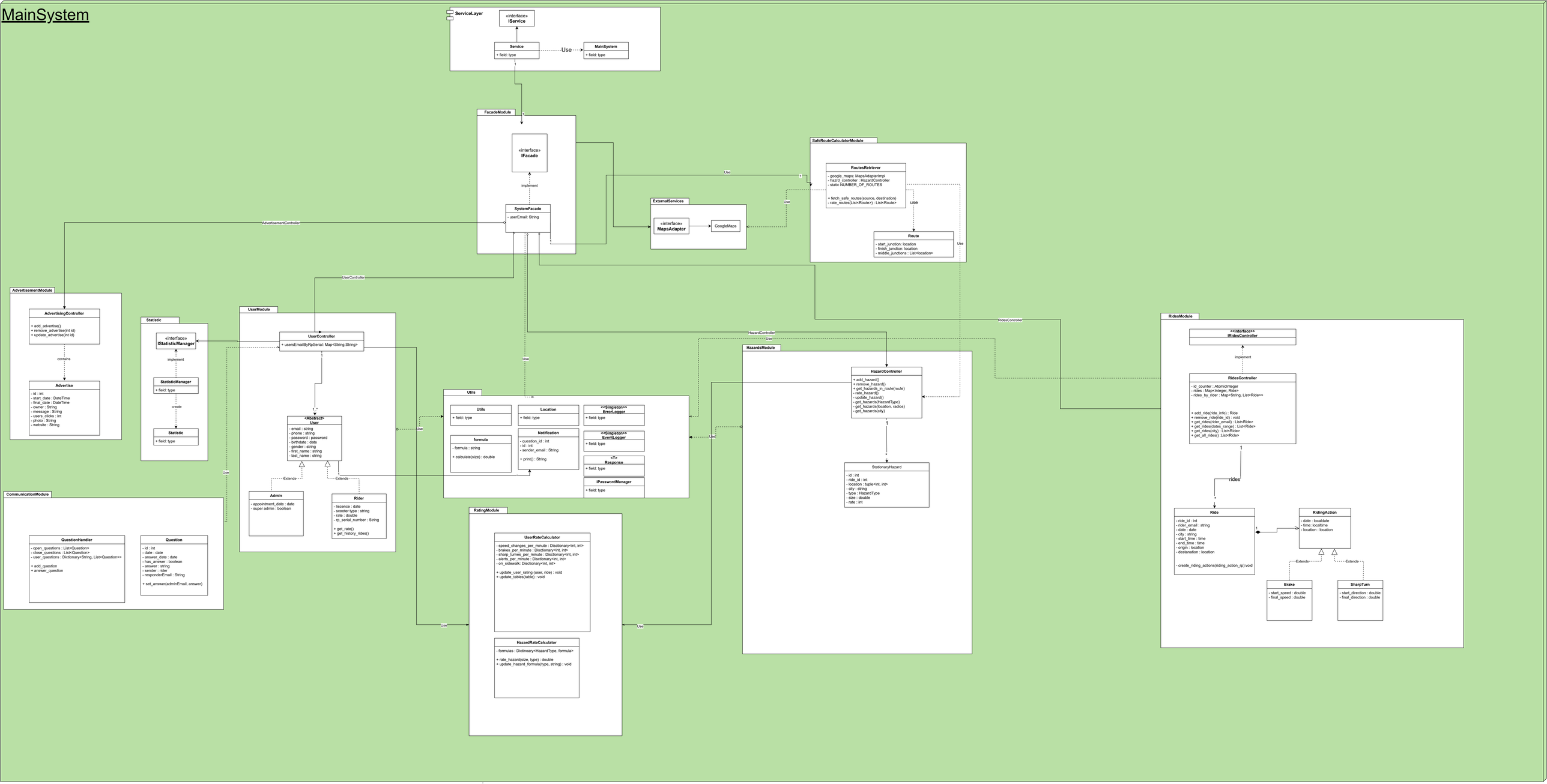

End Ride Sequence
Raspberry pi
service:IService
facade:Facade
rideController:RideController
userController: IUserController
rider: Rider
hazardController: IHazardController
logger: Logger
finish_ride(RideInfo, rp_id)
finish_ride(rideInfo, id)
add_ride(ride_info)
Create(ride_info)
ride:Ride
ride
update_user_rate(rp_id, ride, number_of_rides)
update_rating(ride, number_of_rides)
update_hazards(hazards, ride_id)
loop hazards
if hazard exists
update_hazard(hazard)
else
add_hazard(ride_id, hazard_info)
add_log(message)
Create()
r:Response<T>
r
r

During Ride Sequence




Start Ride Sequence
Scooter Driver
Rider API
service:IService
rideController:RideController
Maps API
GetRoutes(source,dest)
GetRoutes(source,dest)
GetRoutes(source,dest)
GetRoutes(source,dest, maxTime, numOfRoutes)
routes
SortBySafety(routes)
routes
Create(routes)
r:Response<T>
r
r
GetInsturctions(route)
GetInstructions(route)
GetInstructions(route)
GetInstructions(route)
instructions
Create(instructions)
ride:Ride
ride
Create(ride.instructions)
r:Response<T>
r
Show Instructions on screen

MainSystem


ServiceLayer
«interface»
IService
Service
+ field: type
Use
MainSystem
+ field: type
1
FacadeModule
«interface»
IFacade
implement
SystemFacade
- userEmail: String
AdvertisementController
UserController
Use
ExternalServices
«interface»
MapsAdapter
GoogleMaps
SafeRouteCalculatorModule
RoutesRetriever
- google_maps: MapsAdapterImpl
- hazrd_controller : HazardController
- static NUMBER_OF_ROUTES
+ fetch_safe_routes(source, destination)
- rate_routes(List<Route>) : List<Route>
use
Route
- start_junction: location
- finish_junction: location
- middle_junctions : List<location>
RidesController
HazardController
AdvertisementModule
AdvertisingController
+ add_advertise()
+ remove_advertise(int id)
+ update_advertise(int id)
contains
Advertise
- id : int
- start_date : DateTime
- final_date : DateTime
- owner : String
- message : String
- users_clicks : int
- photo : String
- website : String
Statistic
«interface»
IStatisticManager
implement
StatisticManager
+ field: type
create
Statistic
+ field: type
UserModule
UserController
+ usersEmailByRpSerial: Map<String,String>
<Abstract>
User
- email : string
- phone : string
- password : password
- birthdate : date
- gender : string
- first_name : string
- last_name : string
Extends
Admin
- appointment_date : date
- super admin : boolean
Rider
- liscence : date
- scooter type : string
- rate : double
- rp_serial_number : String
+ get_rate()
+ get_history_rides()
Utils
Utils
+ field: type
Location
+ field: type
<<Singleton>>
ErrorLogger
+ field: type
formula
- formula : string
+ calculate(size) : double
Notification
- question_id : int
- id : int
- sender_email : String
+ print() : String
<<Singleton>>
EventLogger
+ field: type
<T>
Response
+ field: type
iPasswordManager
+ field: type
HazardsModule
HazardController
+ add_hazard()
+ remove_hazard()
+ get_hazards_in_route(route)
- rate_hazard()
- update_hazard()
- get_hazards(HazardType)
- get_hazards(location, radios)
- get_hazards(city)
StationaryHazard
- id : int
- ride_id : int
- location : tuple<int, int>
- city : string
- type : HazardType
- size : double
- rate : int
RidesModule
<<interface>>
IRidesController
implement
RidesController
- id_counter : AtomicInteger
- rides : Map<Integer, Ride>
- rides_by_rider : Map<String, List<Ride>>
+ add_ride(ride_info) : Ride
+ remove_ride(ride_id) : void
+ get_rides(rider_email) : List<Ride>
+ get_rides(dates_range) : List<Ride>
+ get_rides(city) : List<Ride>
+ get_all_rides() :List<Ride>
rides
Ride
- ride_id : int
- rider_email : string
- date : date
- city : string
- start_time : time
- end_time : time
- origin : location
- destanation : location
- create_riding_actions(riding_action_rp):void
RidingAction
- date : localdate
- time: localtime
- location : location
Extends
Brake
- start_speed : double
- final_speed : double
SharpTurn
- start_direction : double
- final_direction : double
CommunicationModule
QuestionHandler
- open_questions : List<Question>
- close_questions : List<Question>
- user_questions : Dictionary<String, List<Question>>
+ add_question
+ answer_question
Question
- id : int
- date : date
- answer_date : date
- has_answer : boolean
- answer : string
- sender : rider
- responderEmail : String
+ set_answer(adminEmail, answer)
RatingModule
UserRateCalculator
- speed_changes_per_minute : Dictionary<int, int>
- brakes_per_minute : Dictionary<int, int>
- sharp_turnes_per_minute : Dictionary<int, int>
- alerts_per_minute : Dictionary<int, int>
- on_sidewalk: Dictionary<int, int>
+ update_user_rating (user, ride) : void
+ update_tables(table) : void
HazardRateCalculator
- formulas : Dictinoary<HazardType, formula>
+ rate_hazard(size_type) : double
+ update_hazard_formula(type, string) : void
Use
DataAccessLayer
HibernateUtils
+ field: type
Data Base

RaspberryPiSystem