Personalized Travel Monitor

Università Degli Studi Milano Bicocca

C. BALDI, S. GALIMBERTI, F. OLIVADESE, S. VITALI

Contents

1 Introduction

1.1 The project

This is the text of the project as proposed on the Score website.

More and more applications are available, provided for example by transport operators such as railway companies, metropolitan or regional transportation authorities, that provide real-time information to users when disruptions occur in their networks. Typically, however, these applications provide generic information, in a "one size fits all" approach, where every user receives the same notifications.

The goal of the project is to develop a system that allows users to receive targeted, personalized information only when their routes of interest are affected by the disruptions, and only at the right time (for example, a disruption that occurs in a metro line when the user is out of town is of no interest to the user). The students are expected to identify one or more transport operators of interest, identify the available sources of information concerning disruptions (which might also be user-generated information spread through social networks, possibly through functions available on the application itself), and to create a system that provides users with information that is as accurate, timely and personalized as possible. Students are encouraged to interact with transport operators and potential users of the system to gather their expectations and needs concerning such a system.

The advent of mobile devices, which allow users to receive services while on the move, has naturally led providers of transport services and mobility-related companies to create applications that allow travelers to stay informed about events, especially disruptive ones, that occur along transport networks. However, these applications are typically restricted to monitoring a single network (or the services of a single operator), and/or deliver notifications that are not user- and context-specific. As a consequence, a user of local transportation services (metro/bus/tram lines) will typically receive notifications even when she is not in town, whereas a user of a long-distance service can monitor the situation of a specific service (e.g., a specific long-distance train, or an airplane), but it is not often that she is able to receive real-time "push" notifications when her trip is disrupted.

The goals of the project are the following:

- To design a system that allows travelers to indicate services to be monitored, and to define the kinds of events and disruptions that should be monitored.
- To develop mechanisms that allow users to receive notifications when events of interest occur for the monitored services, according to the preferences and the context of the user (e.g., only if the event is relevant given the current situation of the user, such as her position).

To achieve the goals of the project, an analysis of available sources of information concerning the status of the transport services of interest should be carried out. Given that information concerning the status of a transportation network is typically proprietary, and not always readily available, if the analysis of data sources reveals a difficulty in accessing the required data, then risk-mitigating actions must be considered (e.g., generating synthetic data).

Given the user-centeredness of the whole idea of the disruption notification system, teams tackling the project should make an effort to identify, possibly with the help of user surveys, the kinds of information and the mechanisms and timings of notifications that travelers find more useful.

1.2 A few remarks

The project was carried out with the Agile methodology in mind, although not everything was done the Agile way, we developed incrementally, collected and refined requirements as we went forward with the development. We also made use of some other Agile practices such as User Stories and the Kanban Board. This way of approaching the project proved very interesting and effective, especially because we were our own stakeholders and product owner, since we planned to use the software for ourselves afterwards.

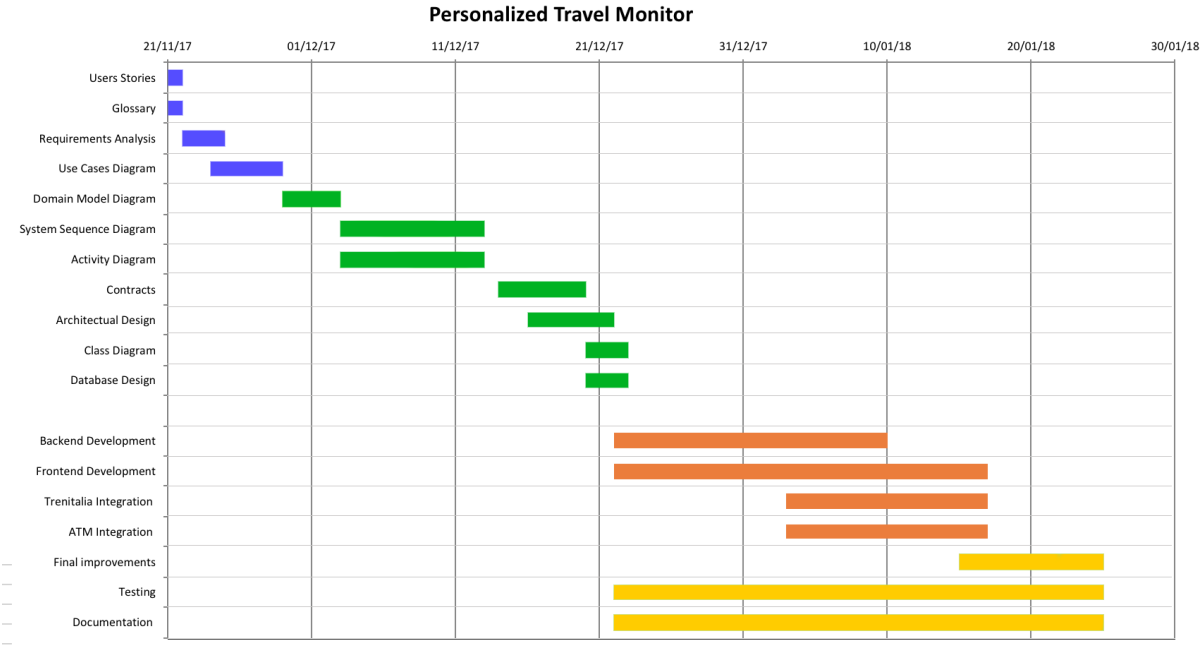
Everthing went mostly according to plan, but in the end, since exams were longer than expected, we decided to not implement some small but not core features (basically a few of the *could* and *want* features detailed in the requirement analysis) but we still carried out the initial analysis for them.

Since we gave the planning phase enough time we were able to design the project very closely to what the implementation turned out to be. This was definitely a great thing for us and it allowed the implementation to be pretty straigth-forward, basically following design documents. The only changes we had to make to the diagrams after the development were only related to naming differences, the application flow was still the same.

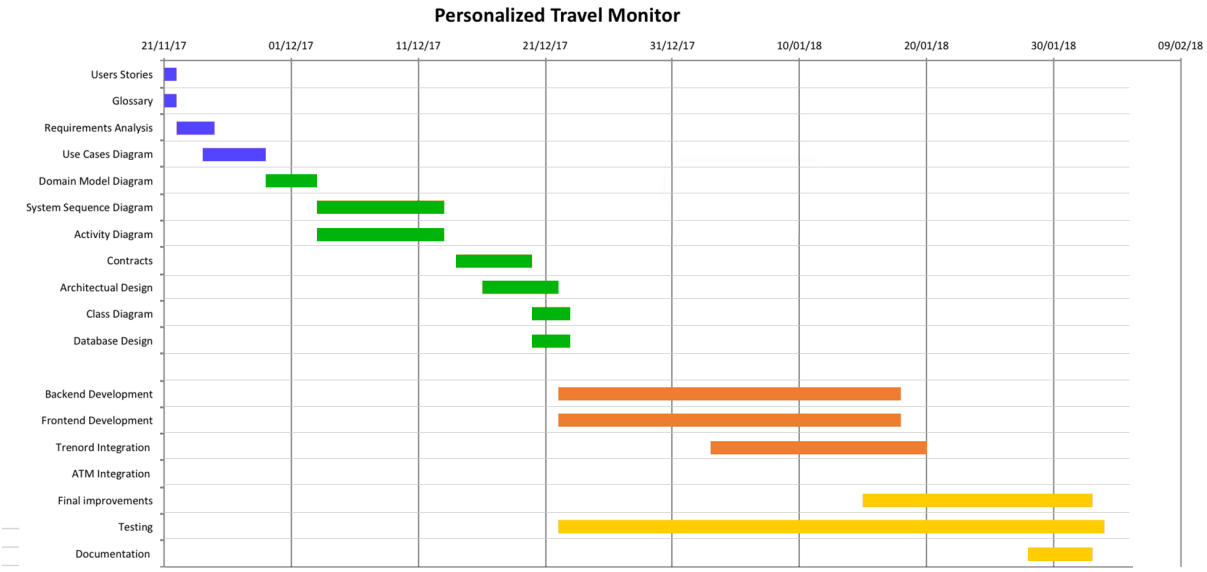
At this moment the only public transportation provider that was implemented is Trenord. The next providers that will be added are Trenitalia and ATM.

1.3 Timeline

Before beginning the project we defined a Gantt chart to plan the evolution of the project.



During the development we updated the Gantt and in the end this was how the project turned out to be.



1.4 Useful Links

- Project Proposal on the SCORE Website
- Repo of the design and analysis documents
- Repo of the code
  - Release Tag: `1.0`
- Kanban Board of the project
- Running instance of the project
  - Use account `demo@demo.com` with password `password` to try out the software; you can also create an account for yourself but this one has a few already inserted trips for testing purpose.

2 Glossary

- **Public Transports**: train, underground, bus and similar means of transportation
- **Traveler**: somebody that uses public transports
- **Trip**: movement from a departure point to a destination with a public transport
  - **Periodic Trip**: Trip repeated several times in a week
  - **Occasional Trip**: trip carried out just one time
- **Unplanned disruption**: delays, cancellations, unexpected changes to a trip
- **Scheduled disruption**: strike or long-term changes in the transport service disclosed by the transportation provider
- **Relevant notification**: Notification relevant for the user's location, line or trip of interest
- **Transport Line**: common route, shared by a set of trips that have same destination and same departure station

3 Requirements Analysis

3.1 User Stories

3.1.1 Commuter

I want to:

- get notified on distruptions regarding public transports of my interest (C1)
- find valid alternatives for my trip in case of a distruption (C2)
- receive real time notifications on unplanned distruptions only if they are relevant to me (C3)
- receive notifications on planned distruptions (C4)
- report unplanned distruptions affecting my trip (C5)
- get periodic statistics about the distruptions that affected me (C6)
- register to the system to be able to use it (C7)
- access the service on every device I own even with low connectivity (C8)
- get notified on distruptions affection the transport lines whenever they are relevant to me (C9)
- share the status of my trip with other people to inform them of the distruptions (C10)
- be able to to communicate with the system's administrator for reporting problems or give suggestions (C11)

3.1.2 Administrator

I want to:

- send notification to the users regarding planned disruptions, technical informations or updates to the system (A1)
- see a report and data about users, trips and usage of the system (A2)
- change/delete user messages to avoid improper use (A3)

3.2 Requirements

3.2.1 Functional requirements

Requirements are in the format of: Requirement [Expected Complexity (1-5), MoSCoW Priority, [Progress]].

Each group of requirements is related to a user story, indicated between round brackets.

(C1)

- Allow the user to follow trips *[4, M, Completed]*
  - To follow trips on a one-time basis *[3, M, Completed]*
  - To follow trips periodically *[4, S, Completed]*
    - The user can specify days of the week in which to follow a trip (by default from Monday to Friday)
- Allow the user to stop following a trip *[2, M, Completed]*
- Send notifications to the user based to distruptions affecting trips he/she is following *[2/3, M, Completed]*

(C2)

- Automatically find alternatives to a given trip affected by distruptions *[5, Missing]*
  - Find alternative public transport solutions
  - Find alternative private transport solutions

(C3)

- Allow the user to configure which notifications he/she should receive *[3, W, Missing]*
  - According to the severity of the issue
  - According to the type of the issue
  - According to the place where the user is located *[5, W, Missing]*
- Allow the user to configure how many notifications he/she should receive *[3, C, Missing]*
  - Frequency
  - Maximum number

(C4)

- Recognize planned distruptions of the service and send notifications to the users at least two days before the event (or as soon as possible if the event is happening in less than two days) *[5, S, Partial]*

(C5)

- Allow the user to report unplanned distruptions involving his trip *[3, S, Completed]*

  - The reports will be divided in categories (Delay, Suppression, Other)
  - It will be possibile to also add an optional short description
- Allow the user to read other's (and his) report involving transport lines of his interest *[2, S, Completed]*

(C6)

- Store statistics about distruptions on trips followed by users *[3, C, Partial]*
  - Average Delay
- Allow to the user to read statistics on the distruptions *[2, C, Completed]*
  - On his trips
  - On his transport lines of interest

(C7)

- Allow the user to register to the system using username, email and password *[3, M, Completed]*
- Allow the user to register using external authentication services (Google, Facebook, etc) *[4, C, Completed]*

(C9)

- Recognize distruptions involving a whole transport line (when possible) in the moments before the trip *[4/5, S, Partial]*
- Send notifications on issues involving a whole transport line *[2/3, S, Missing]*

(C10)

- Allow the user to share the status of a trip with external services (Facebook, Twitter, Email, Whatsapp) *[2, C/W, Completed]*

(C11)

- Allow the user to contact the administrator through email *[1, S, Completed]*

(A1)

- Allow the administrator to send textual notification to the users *[2/3, C, Completed]*

(A2)

- Allow the administrator to see data about the state of the system, users, trips in a specific section *[3, C, Completed]*

(A3)

- Allow the administrator to delete user's report*[3, S, Completed]*

3.2.2 Non-functional requirements

Interoperability

- APIs for external and internal use will be provided by the system, allowing for all the user-functionalities to be performed programmatically, also allowing external integration

Performance

- Discover distruptions on public transport within 4 minutes of their appearence
- The service should load (first paint load) in less than 2 seconds
- The service should load relevant data in less than 5 seconds

Scalability

- The system should be as decentralized as possible to allow for multiple information-retrieval nodes to be used
- The user will be able to follow at max 10 trips per day

Capacity

- The system should be able to handle 1000 users and 5000 daily unique trips

Availability

- The system should try to keep 24/7 uptime
  - The system should grant service between 05:00 and 24:00
- We can't grant full availability if external services are down

Usability

- Allow the user to access the service from a smartphone, tablet, computer while it is connected to the internet

Recoverability

- The system should gracefully handle failures of external services without crashing
- If the system crashes it will automatically restart

Maintainability

- If there are critical bugs they will be fixed whithin 3 working days
- If there are non-critical bugs they will be fixed whithin 2 working weeks
- Update will be performed in regular intervals

Security

- User data will be kept safe and private
  - Passwords will be at least 6 character long
- The website will not allow for unauthorized access
- The website will not allow for injections and XSS

Localization

- The system will be in English but will be designed with future support of other languages

Data Integrity

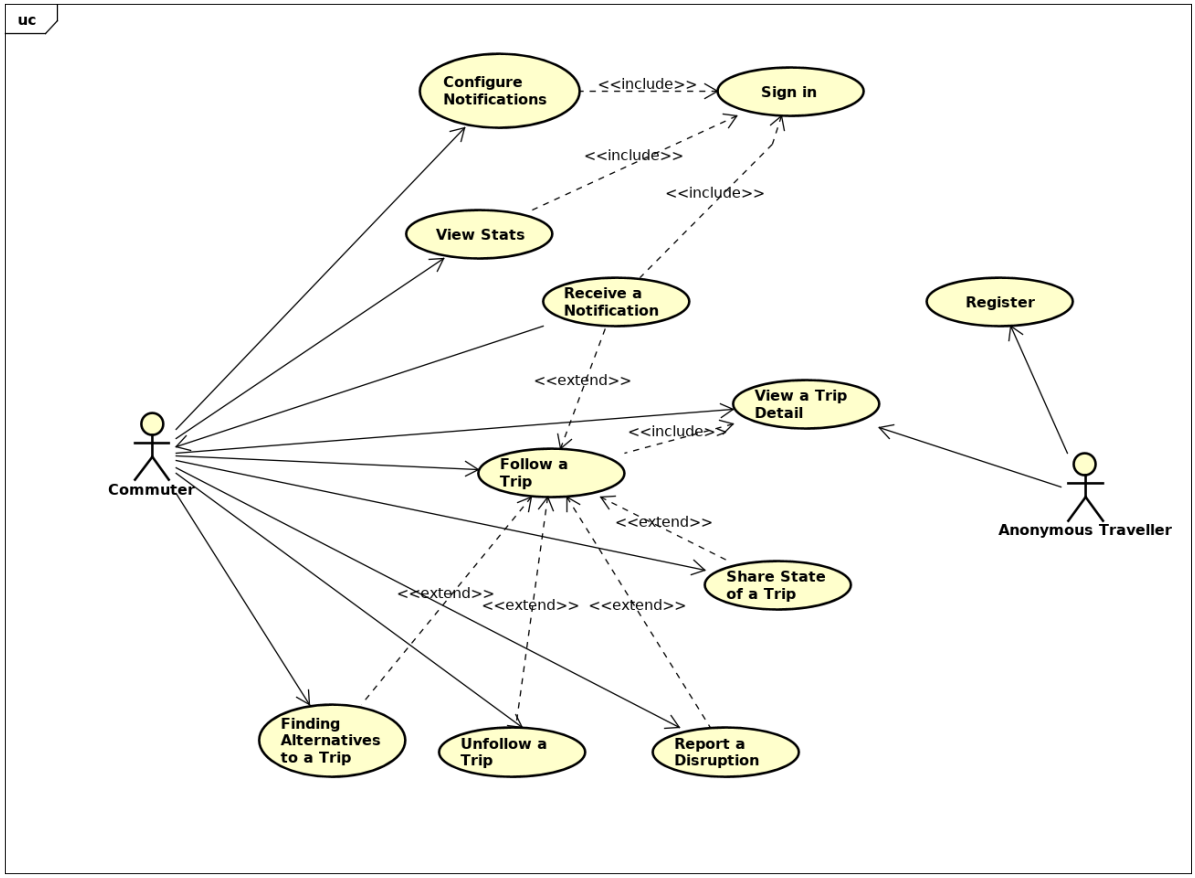
- Data will be stored in the database as needed
  - Detailed statistics for the current month will be saved
  - Summarized data will be stored for the previous months
- Unneeded data will be deleted as soon as it is not needed

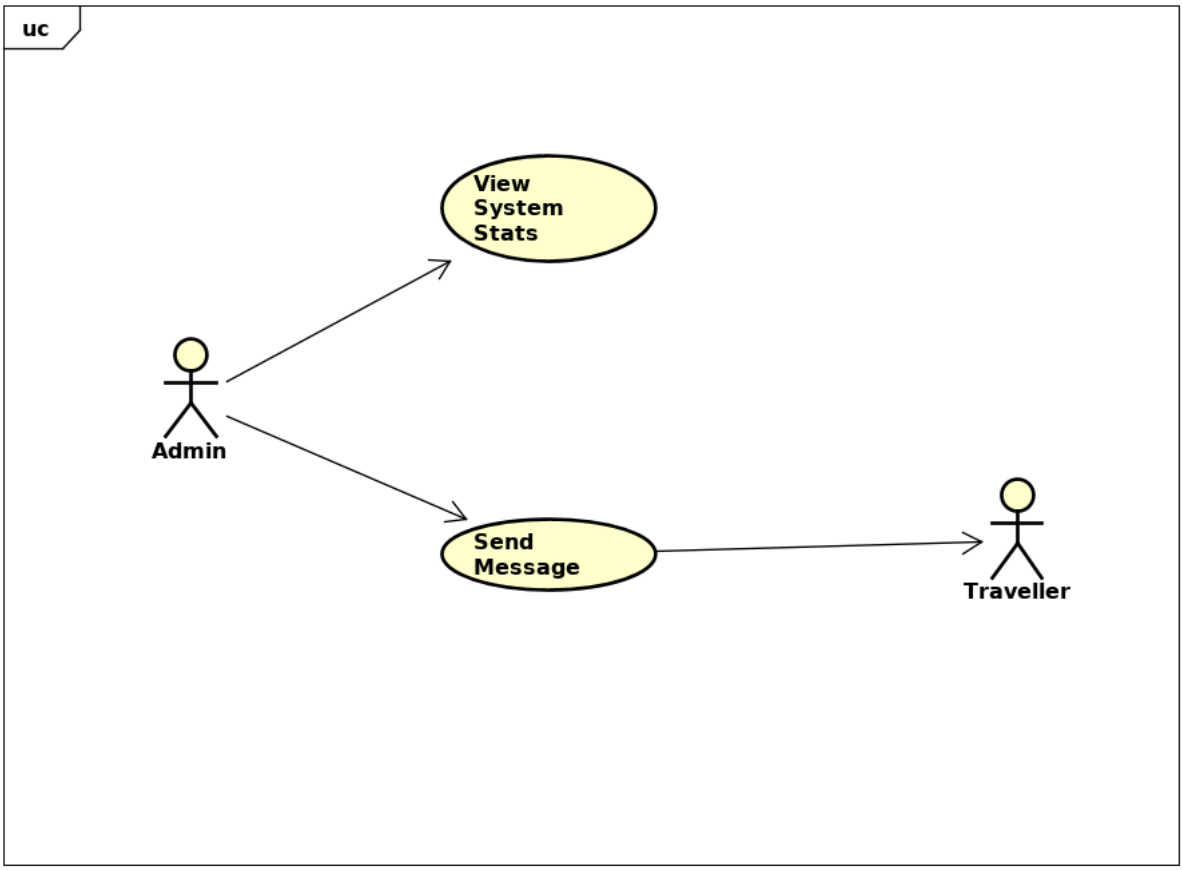
3.2.3 Domain requirements

- Minimize the number and the frequency of requests to external services both keeping performance and rate-limiting in mind
- Follow design and implementation standards (W3C, Google)

3.3 Use Cases

3.3.1 Diagrams





3.3.2 Anonymous Traveler

Registering an account

- The user connects to the system
- The user chooses to register
- The users inserts an username, an email and a password
- The system confirms that the registration has been performed
- The system redirects the user to the starting page

Alternative scenarios

- The user chooses an username or an email that was already taken by someone else. The system will tell the user that he must repeat the registration with different information
- The user inserts a password that doesn't comply to security standard. The system will tell the user that he must repeat the registration with different information
- The user inserts a malformed email. The system will tell the user that he must repeat the

registration with different information
- The user doesn't insert one of the required fields. The system will tell the user that he must repeat the registration with all the required information

View a Trip Detail

- The user connects to the system
- The user chooses to see the details of a trip
- The user chooses the vehicle type
- The user chooses departing and arrival location and time
- The user will see the entire list of compatible solutions found with their details

Alternative Scenarios

- No compatible solutions were found, an error message will be shown
- The user inserts incomplete data; no compatible solutions will be found

3.3.3 Traveller

Signin in an account

- The user connects to the system
- The system decides to log-in
- The user inserts the required data
- The system will check the data and redirect the user to the starting page

Alternative scenarios

- The user inserts wrong information; the system will show an error message and ask the user to repeat the operation
- The user is already logged in; the system will recognize him and will redirect him to the starting page

Follow a trip

- The user connects and logs-in into the system
- The user chooses the vehicle type
- The user chooses departing and arrival location and time
- The user will see the entire list of compatible solutions found
- The user selects one of the solutions proposed

- The user confirms the choice

Alternative scenarios

- If no compatible solutions are found an error message will be shown to the user
- The user doesn't confirm the selection, the operation will be discarded
- The user doesn't insert all the required informations and therefore no solutions will researched
- The user specifies that the trip is recurring on multiple days; the user will insert the days in which the trip should be monitored

Finding alternatives to my trip

- The user connects and logs-in into the system
- The user selects a trip from the list of trips that he's following
- The user chooses to select alternative to that trip inserting similar information
- The system shows a list of compatible solutions found
- The user chooses one of the solutions

Alternative scenarios

- If no compatible solutions are found an error message will be shown to the user

Report a Distruption

- The logged-in user chooses a trip of interest
- The user chooses to report an information
- The user inserts some information and confirms his intentio to send the message
- The system receives and shows to everyone interested in the same trip the user's report

Alternative scenarios

- The user doesn't give a confirmation; the report will be discarded
- The user wants to send multiple report and therefore will perform this operation multiple times

Share the state of the trip

- The user connects and logs-in into the system
- The user selects a trip from the list of the trips that is following
- The user chooses to share the state of the trip using an external sharing service

- The system performs the required operation and redirects the user the user to the trips'detail page

Aternative scenarios

- The user isn't following any trip; he must register a trip before sharing its state

View users stats

- The user connects and logs-in into the system
- The user chooses to see his personal statistics
- The system shows information and report that regards the user

Unfollow a trip

- The user connects to the system
- The user selects, from the list of the trips followed by himself, the one that doesn't want to follow anymore
- The user confirms the selection

Alternative scenarios

- There are no trips followed by the user, the operation is not possible

Receive a notification

- The user's device shows to the user the notification sent by the system

Contact the administrator

- The users sends an email to the administrator

Configure notification frequency and type

- The user connects to the system
- The user selects the frequency of the notifications that will be sent to him
- The user chooses the type of notification that matters to him
- The user confirms the selection
- The system confirms that the modifications took places

Alternative scenarios

- If the change is not confirmed the system will keep the last settings

3.3.4 Admin

Manually send a notification

- The admin connects to the system
- The admin chooses the type of notification to send
- The admin compile the form for notifications and confirm the operation
- The admin chooses the users to send the notification to (all by default)
- The system send the notification to the users

Alternative scenarios

- The admin inserts wrong information; the system will show an error message and ask the user to repeat the operation
- The admin is already logged in; the system will recognize him and will redirect him to the starting page
- If the admin don't compile all the fields of the form for the notification, the system reports the error
- If the notification send fails the system reports the error with an alert.

View detailed stats

- The admin connects to the system
- The admin chooses to see the system statistics
- The admin chooses to see the users statistics
- The admin chooses to see the programmed trips statistics

3.4 Detailed Use Cases

3.4.1 Traveller - Following a trip

- **Range**: User Application
- **Level**: User scope
- **Priority**: Highest
- **Main Actor**: Commuter, the user of the service that wants to start receiving pieces of information regarding 1 trip of interest
- **Pre-Conditions**: The user has logged-in in the system
- **Post-Conditions**: The service correctly registered the interest of the user regarding a trip and has started monitorig it
- **Frequency of use**: no more than 5 times per user

Other:

- **Technological variants**: nothing
- **I/O variants**: nothing

Main Scenario

- The user connects to the system
- The user logs in
- The user decides to register a new trip to be monitored
- The user chooses the type of vehicle that wants to use
- The user inserts the departing location of the trip
- The user inserts the destination of the trip
- The user inserts the day and the time of the trip
- The system shows to the user a list of the trips compatible with the data given in input
- The user selects a trip from the list
- The user confirms the selection
- The system confirms to the user that the subscription has been completed

Extensions

Alternative success scenario: Already logged-in

- The user connects to the system
- The user is already logged in and therefore recognized by the system
- same operations as the main scenario

Alternative success scenario: Recourring Trip

- same operations as the main scenario until point 9
- The user selects a trip from the list
- The user specifies that the trip is recurring
- The user selects from a list the days in which the trip should be monitored
- The user confirms the choice
- The system confirms to the user that the subscription has been completed

Alternative failure scenario: Unrecognized station

- same operations as the main scenario until point 4
- The user inserts the departing location of the trip
- The user inserts the destination of the trip
- The system doesn't find a compatible solution with the data given in input the; an error message will be shown to the user

Alternative failure scenario: No available trip

- same operations as the main scenario until point 6
- The user inserts the day and the time of the trip
- The system doesn't find a compatible solution with the data given in input the; an error message will be shown to the user

Alternative failure scenario: No confirmation

- same operations as the main scenario until point 9
- The user selects a trip from the list
- The user doesn't confirm the operation that will be discarded

Special Requirements

Performance

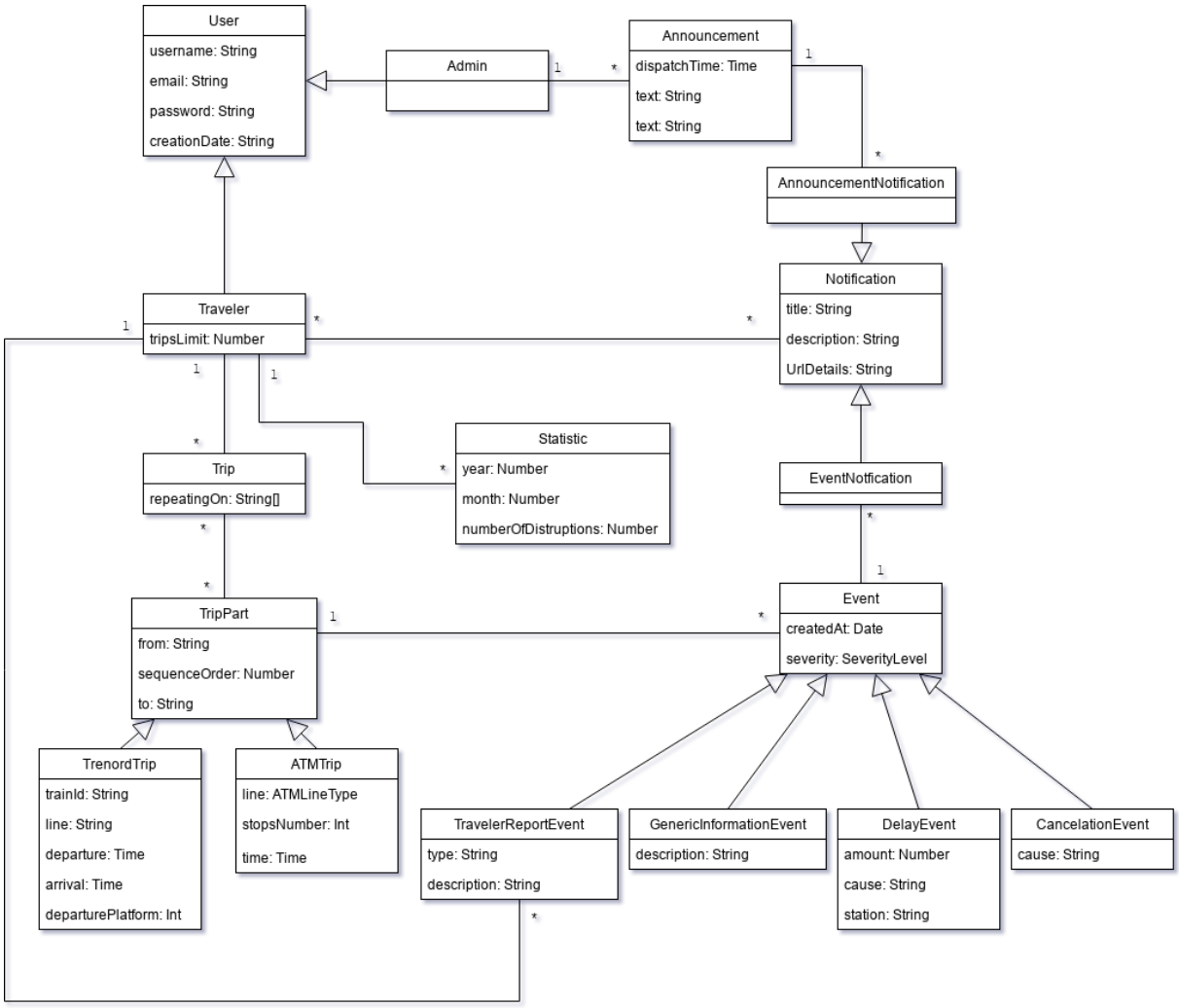
- The system must show the list of compatible solutions in less than 10 seconds
- The system should detect the presence of errors in the data inserted by the user in less than 5 seconds

Usability

- The user interface must be easily understandable
- The interface must adapt to different types of devices (smartphone, tablet and desktop)

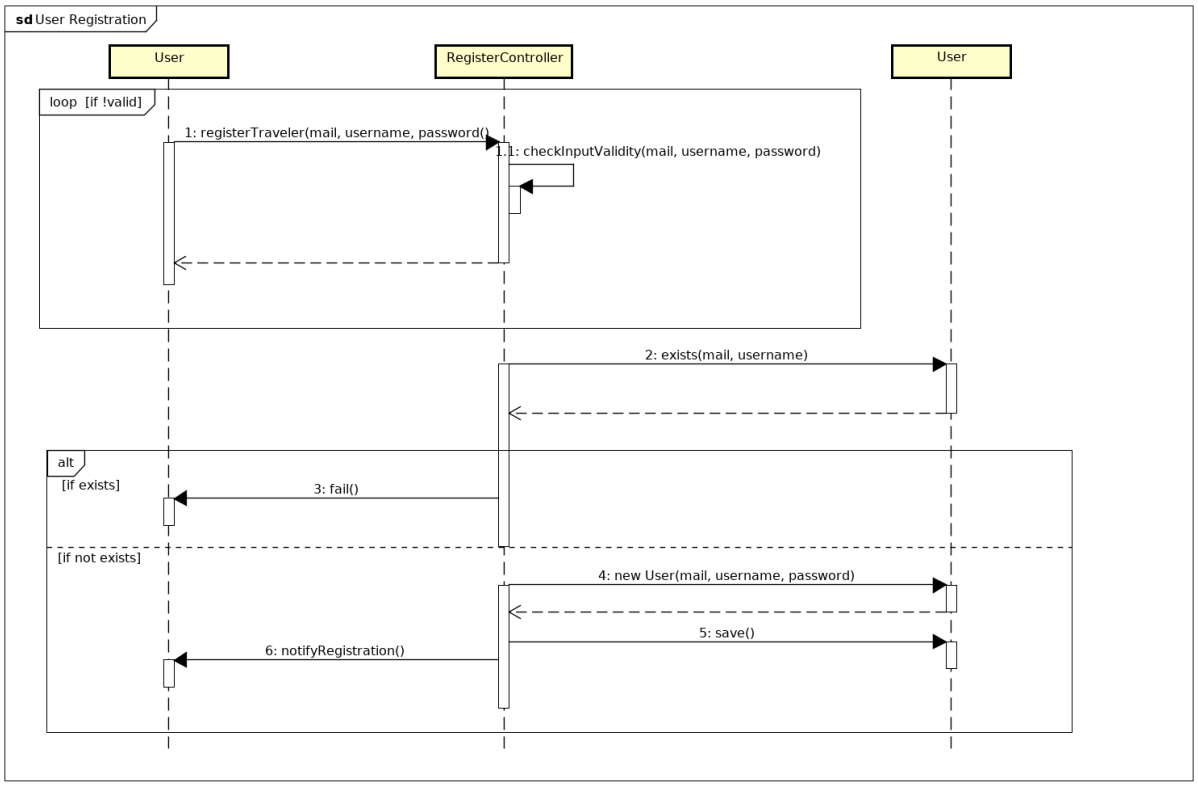
4 Design

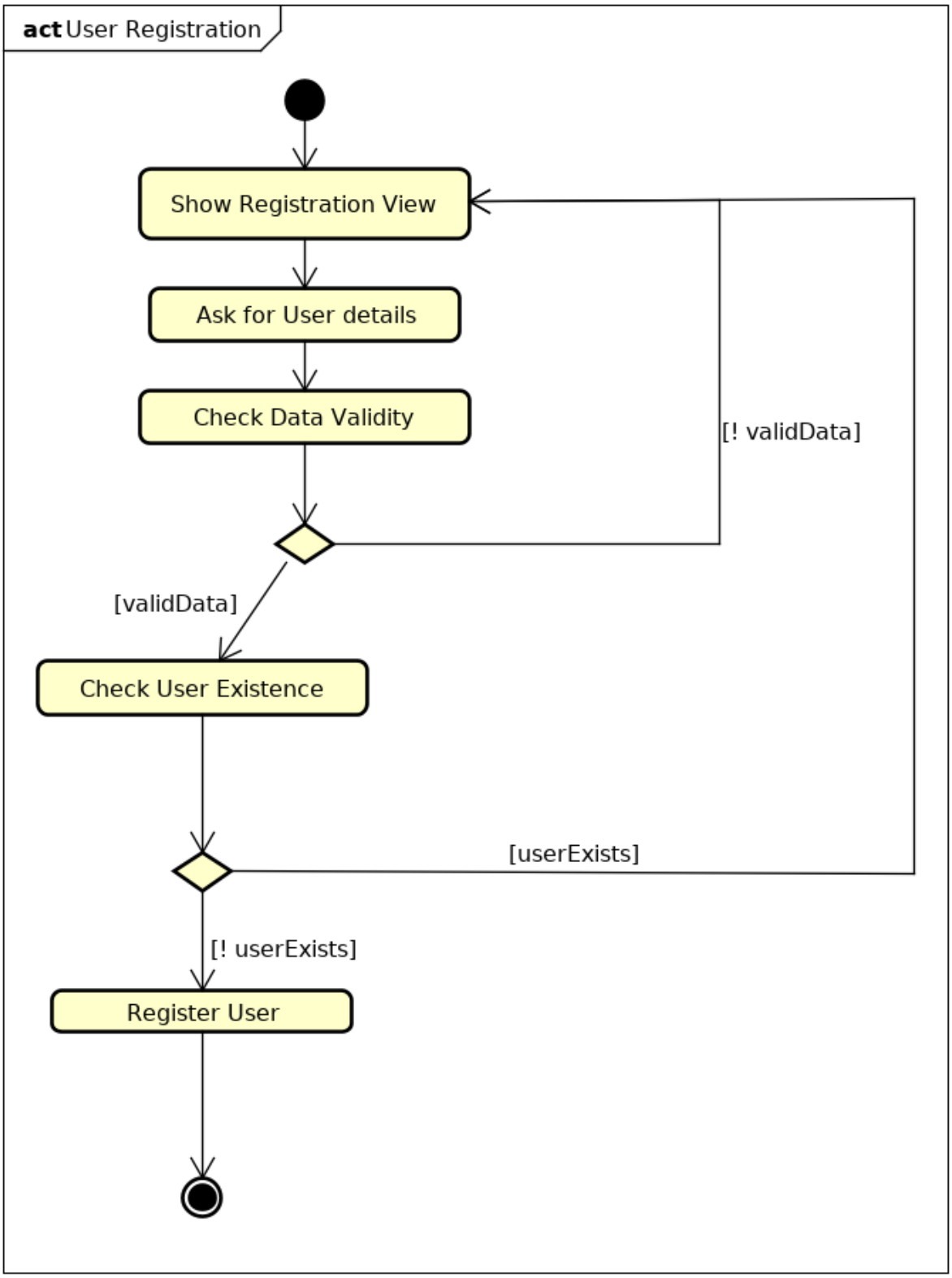
4.1 Domain Model Diagram



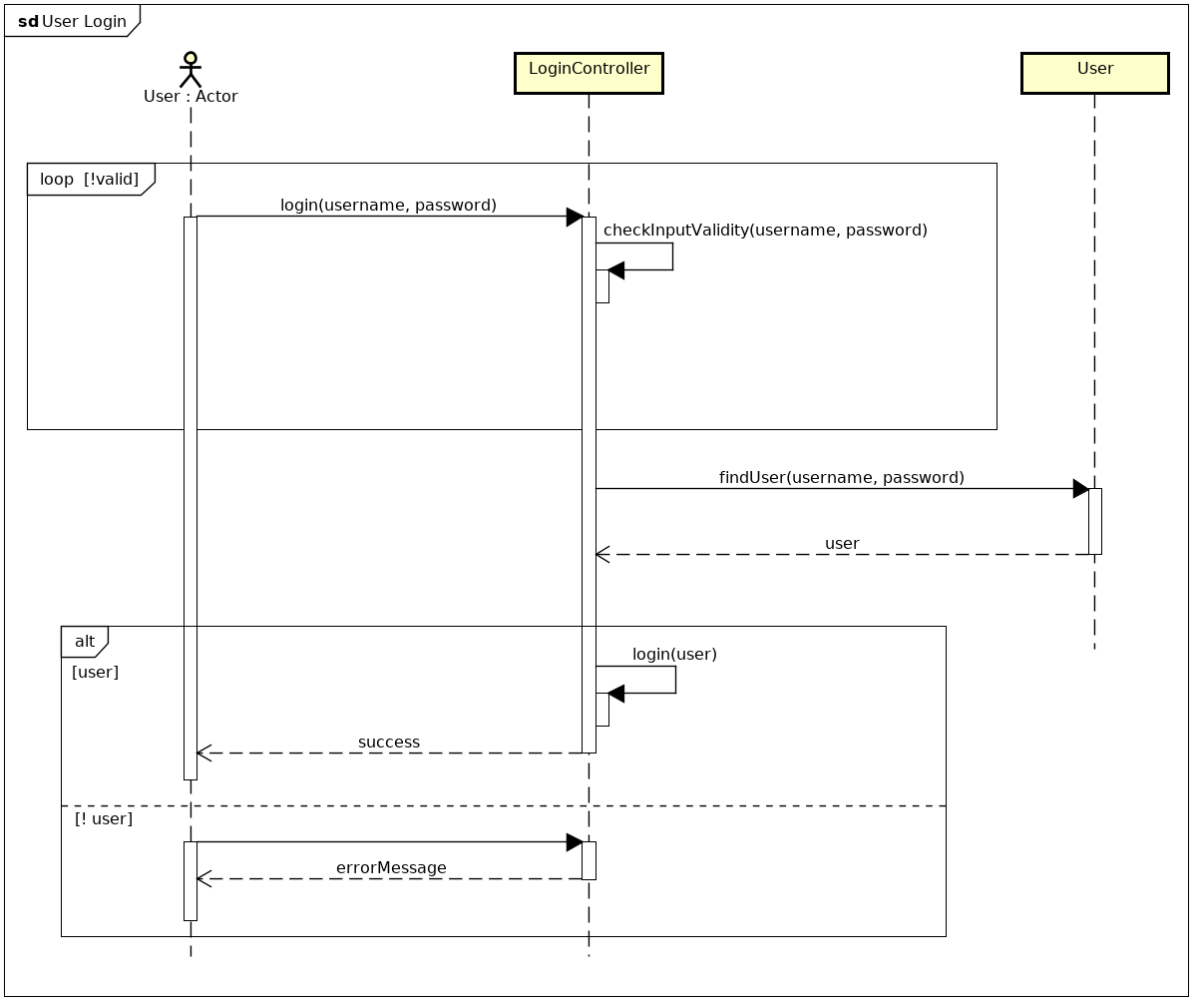
4.2 Sequence and Activity Diagrams

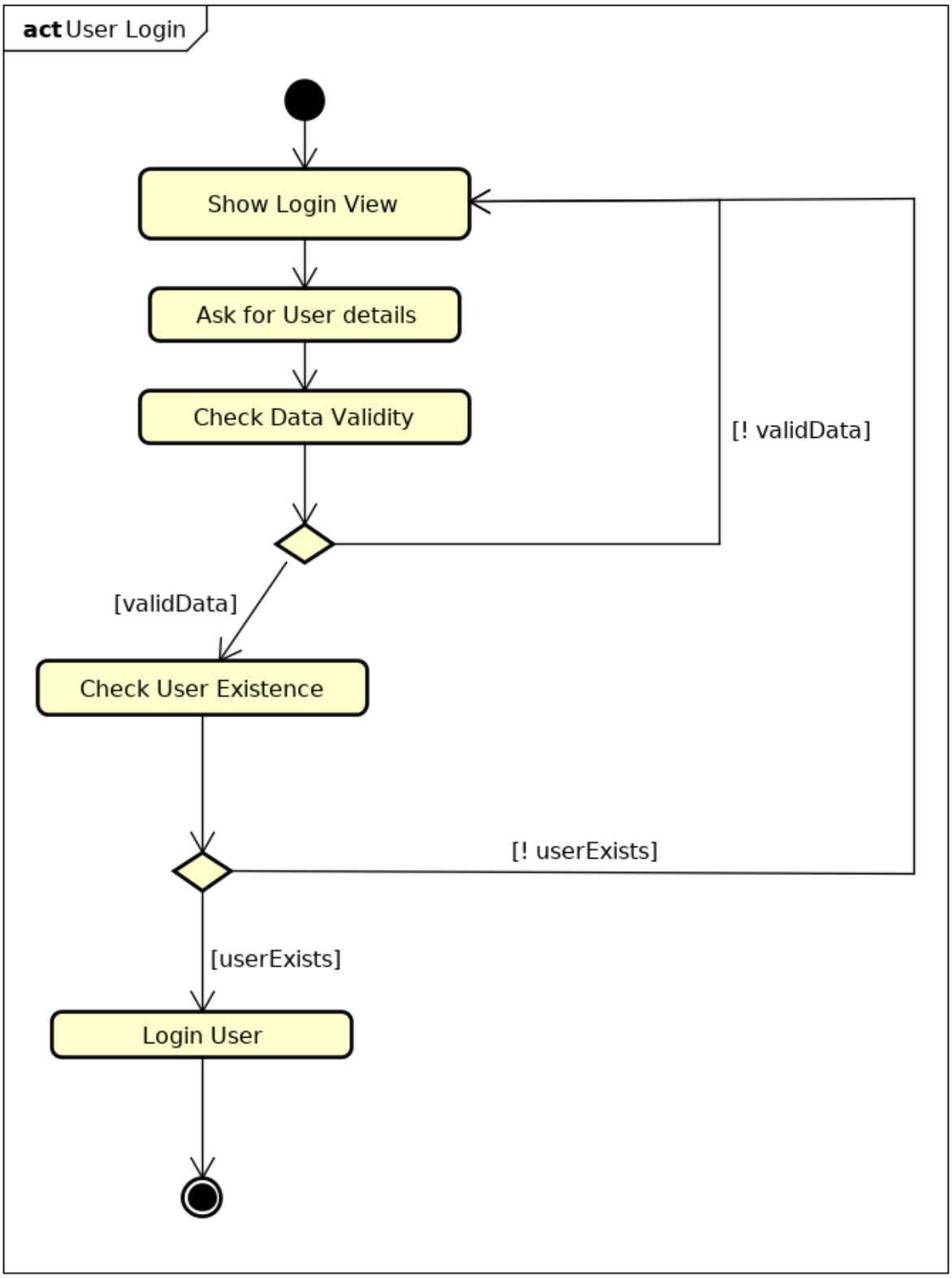
4.2.1 User Registration



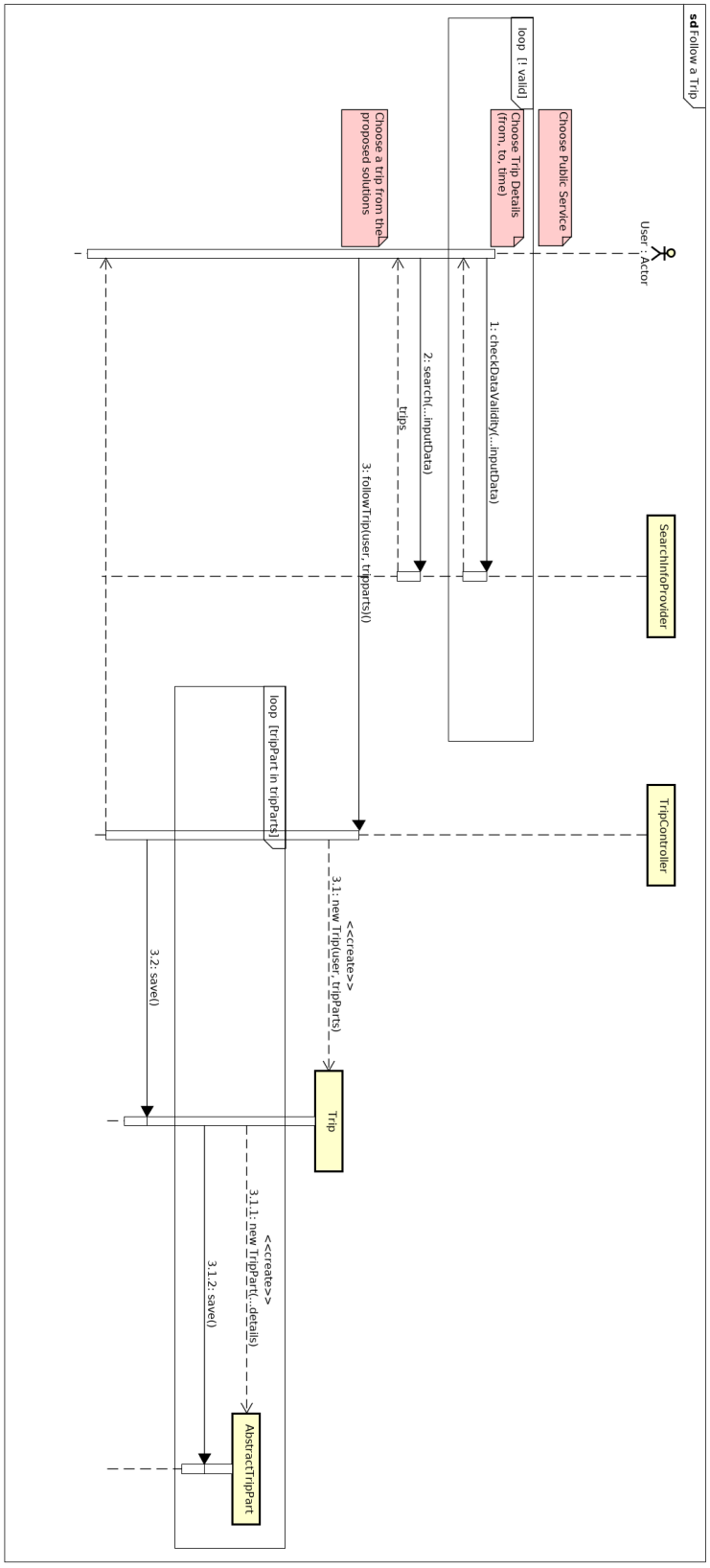

act User Registration
Show Registration View
Ask for User details
Check Data Validity
[! validData]
[validData]
Check User Existence
[userExists]
[! userExists]
Register User

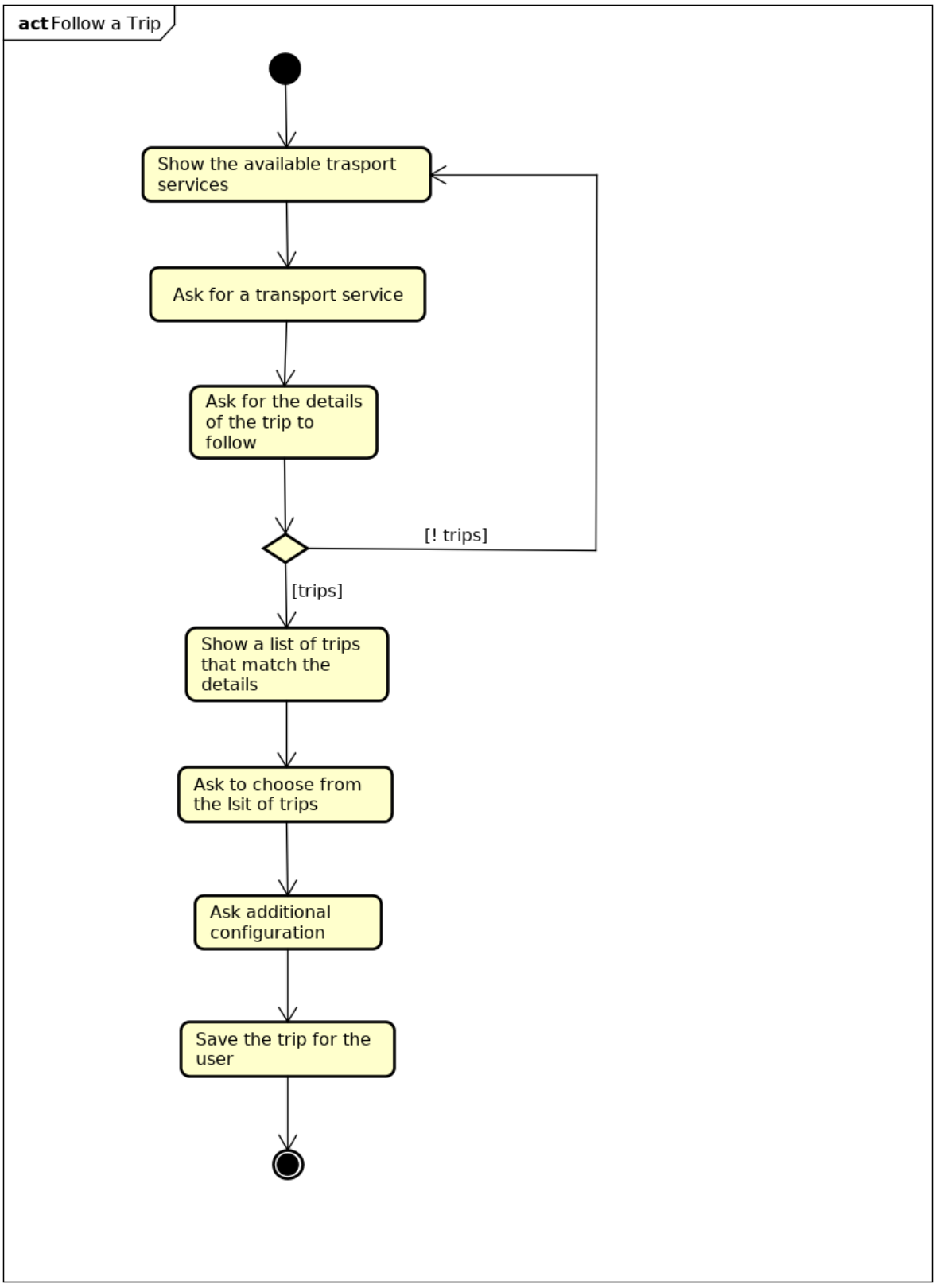
4.2.2 User Login



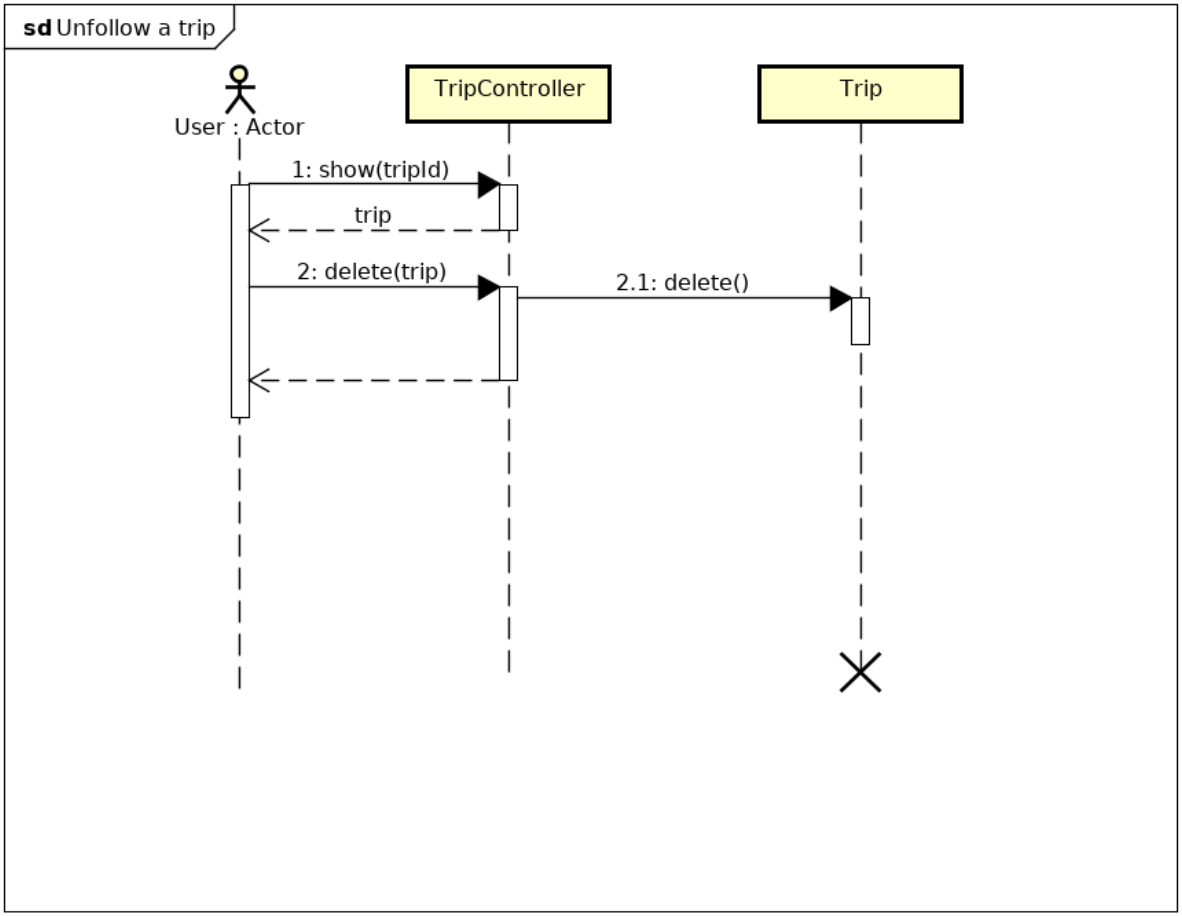

act User Login
Show Login View
Ask for User details
Check Data Validity
[! validData]
[validData]
Check User Existence
[! userExists]
[userExists]
Login User

4.2.3 Follow A Trip

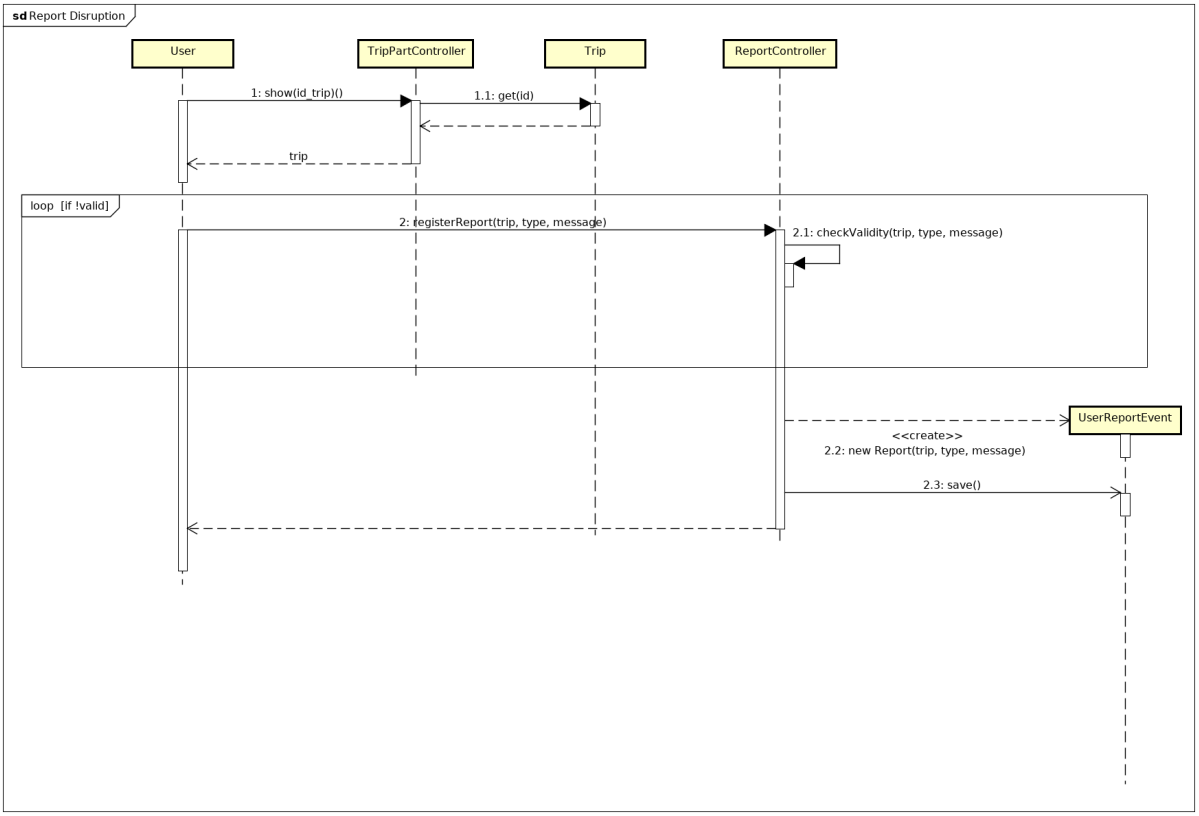


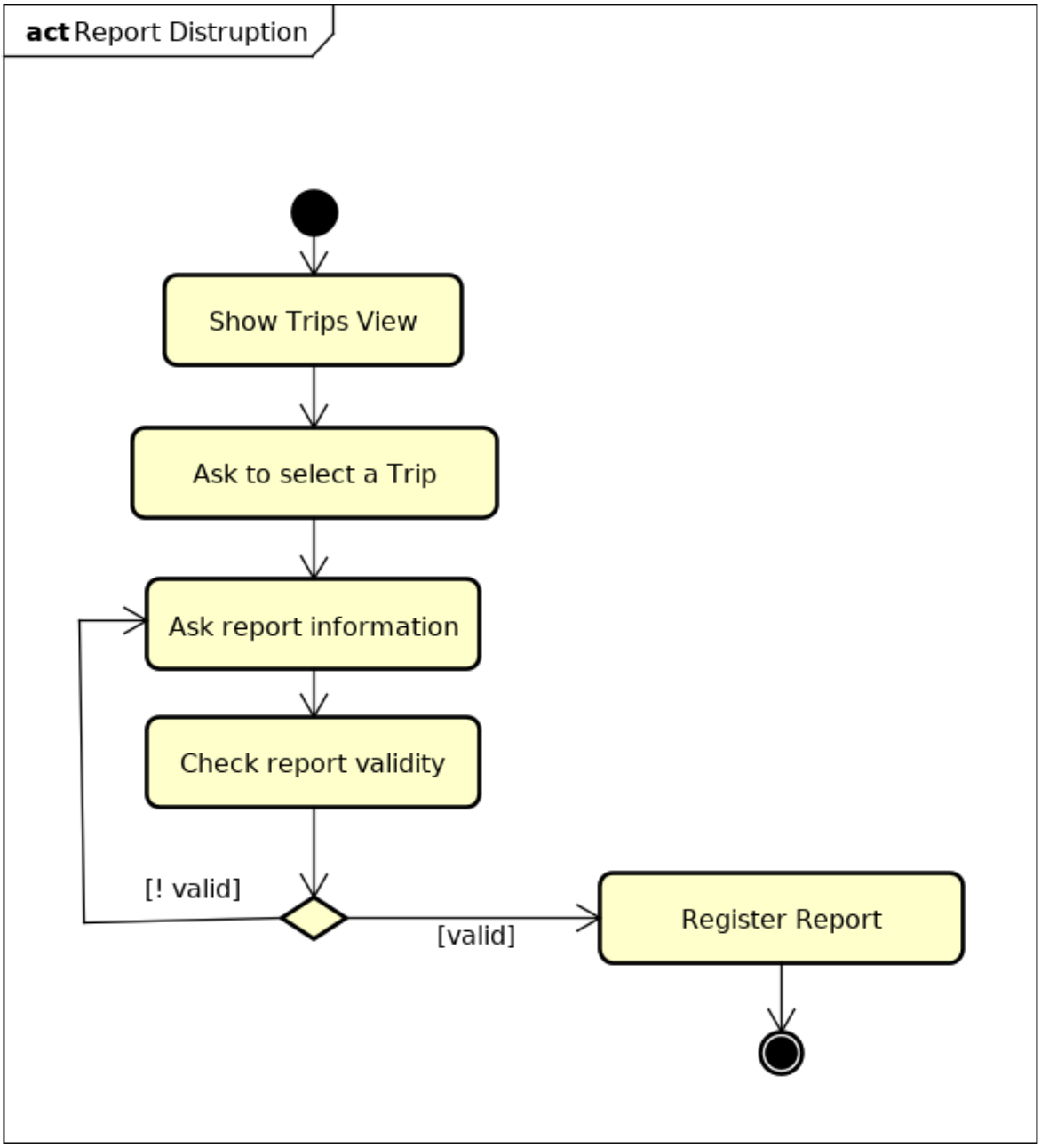

act Follow a Trip
Show the available trasport services
Ask for a transport service
Ask for the details of the trip to follow
[! trips]
[trips]
Show a list of trips that match the details
Ask to choose from the lsit of trips
Ask additional configuration
Save the trip for the user

4.2.4 Unfollow A Trip

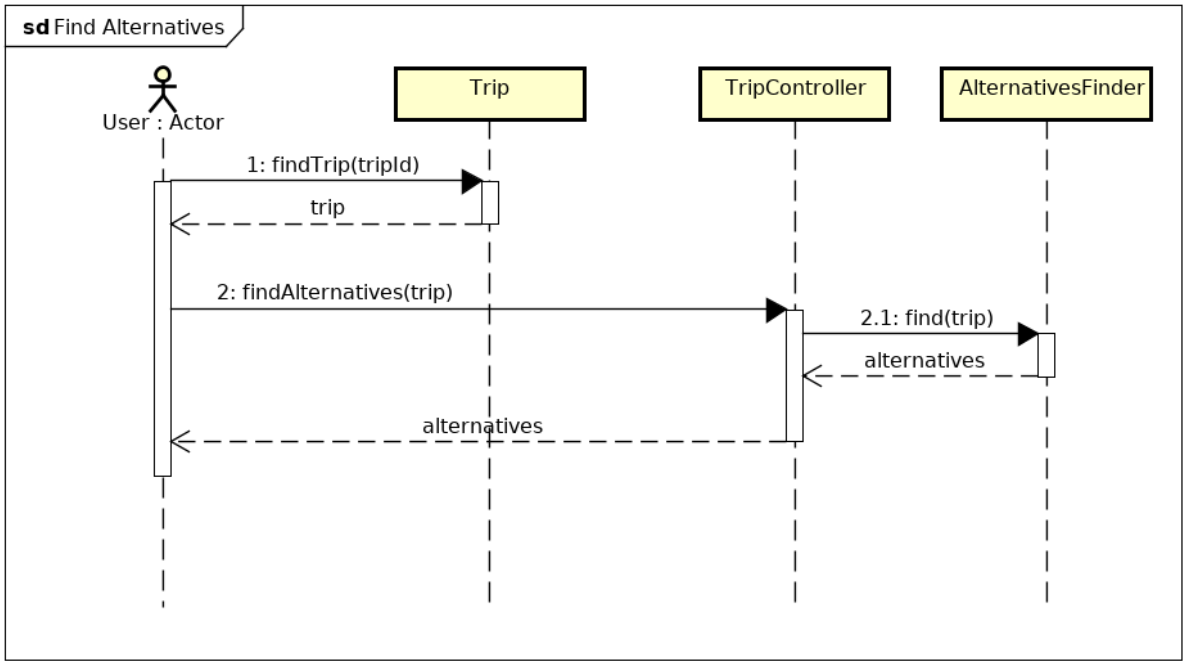


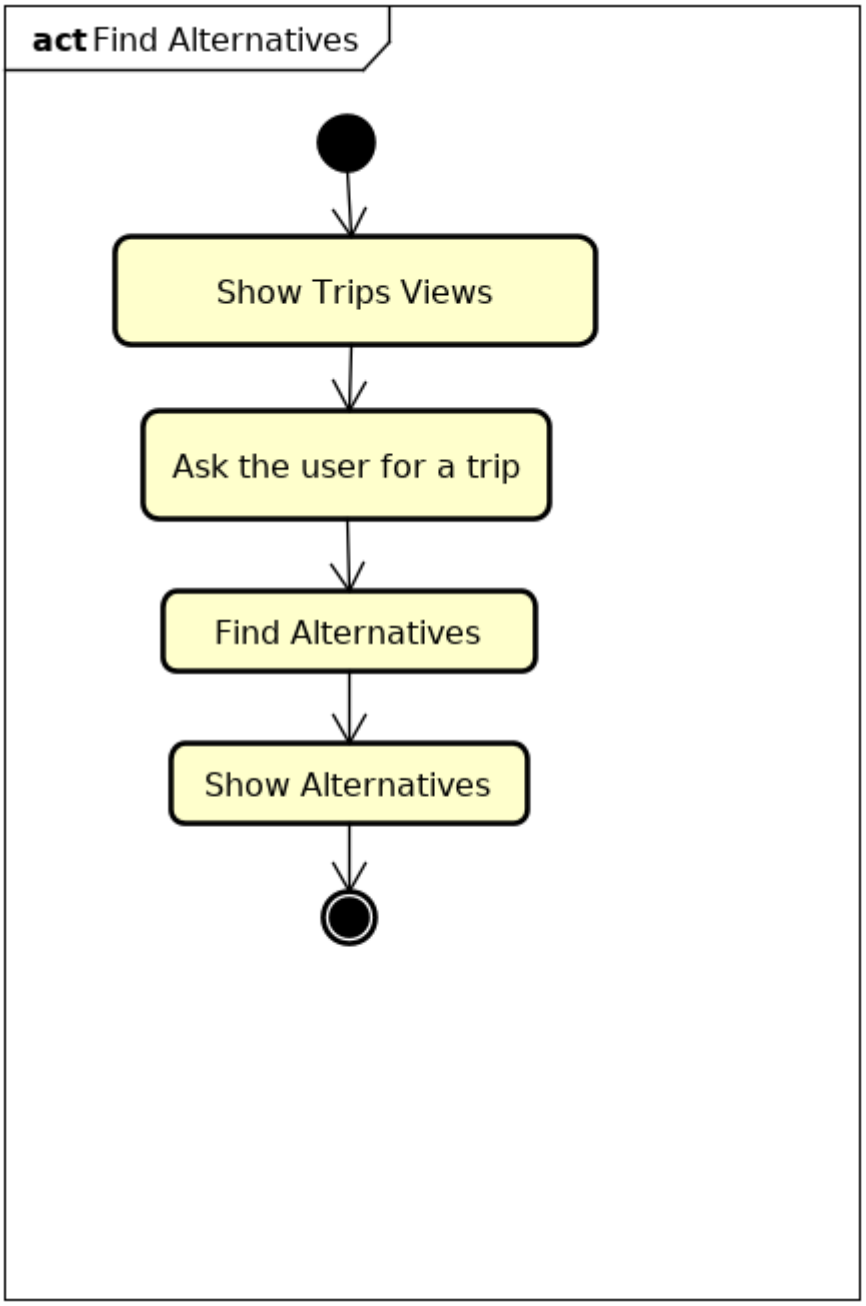
4.2.5 Report Distruption



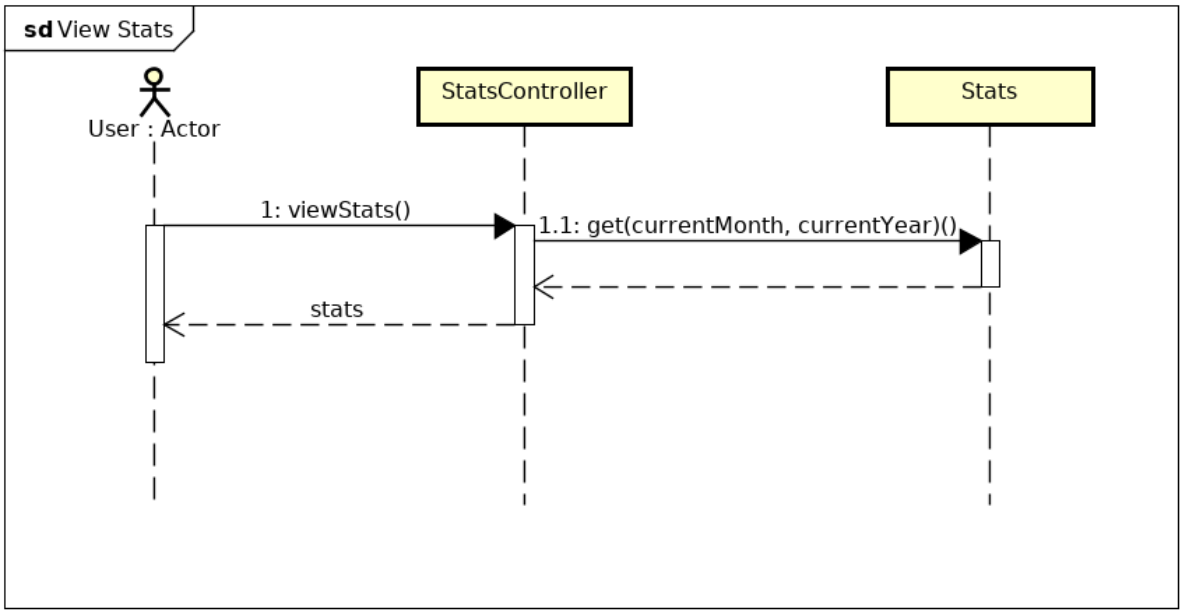

act Report Distruption
Show Trips View
Ask to select a Trip
Ask report information
Check report validity
[! valid]
[valid]
Register Report

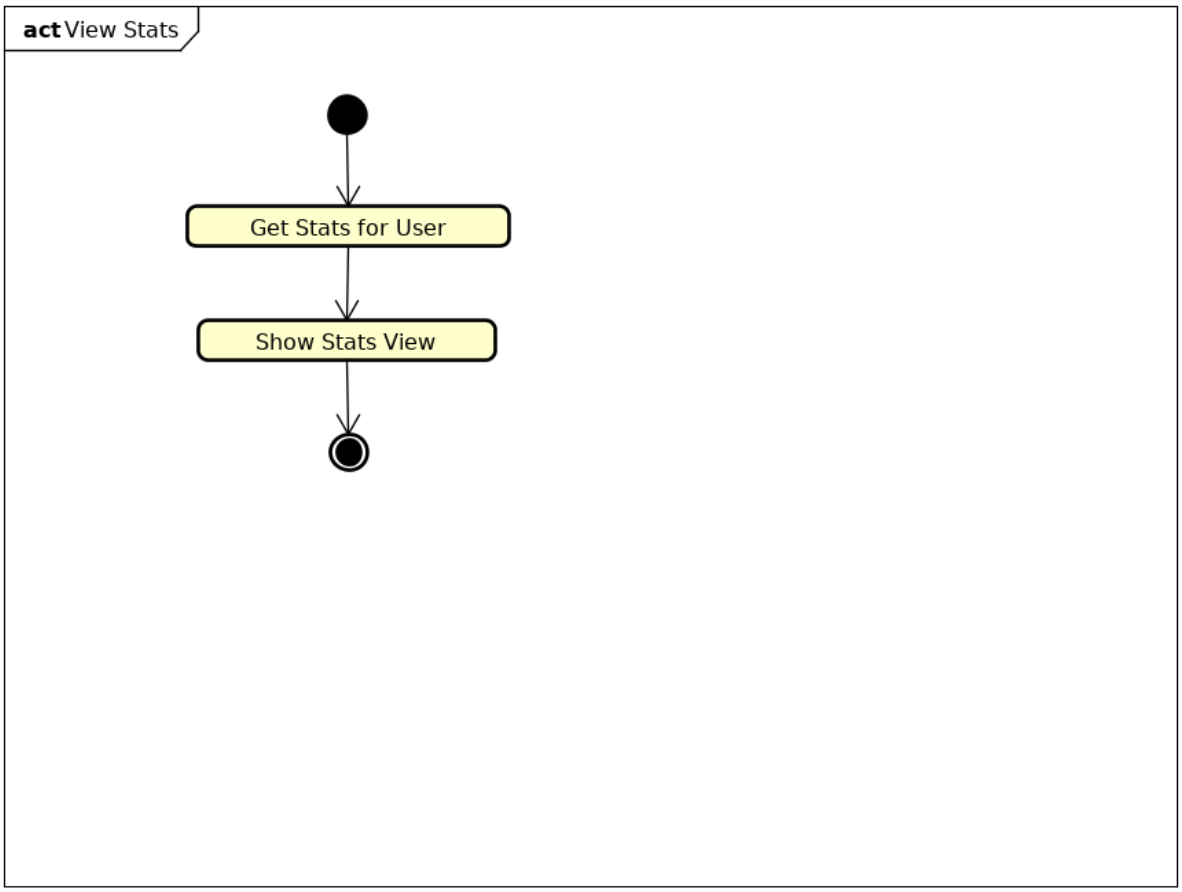
4.2.6 Find Alternatives



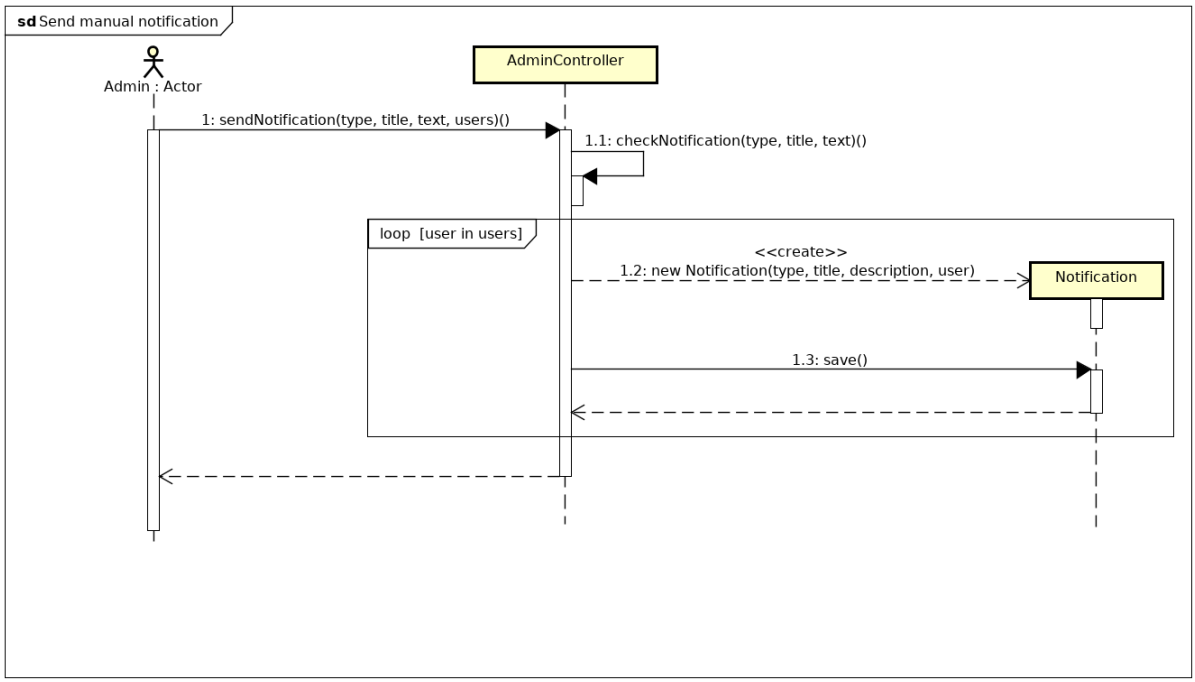

act Find Alternatives
Show Trips Views
Ask the user for a trip
Find Alternatives
Show Alternatives

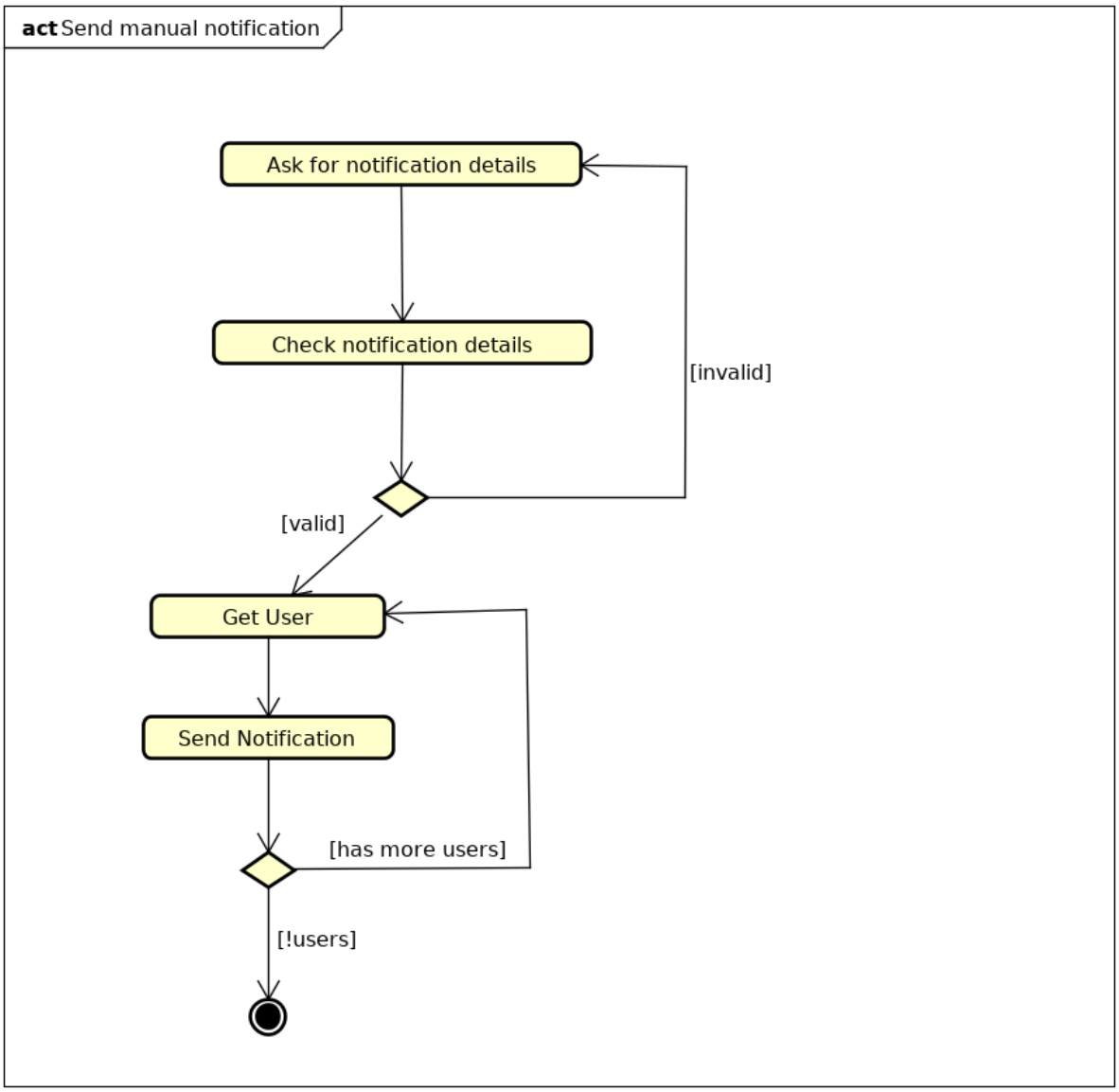
4.2.7 View Stats





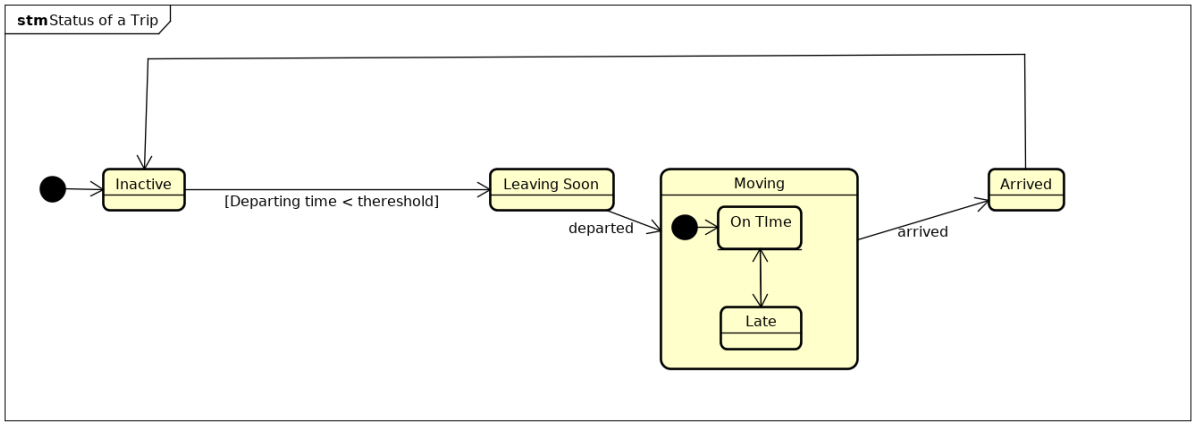
4.2.8 Send Manual Notification



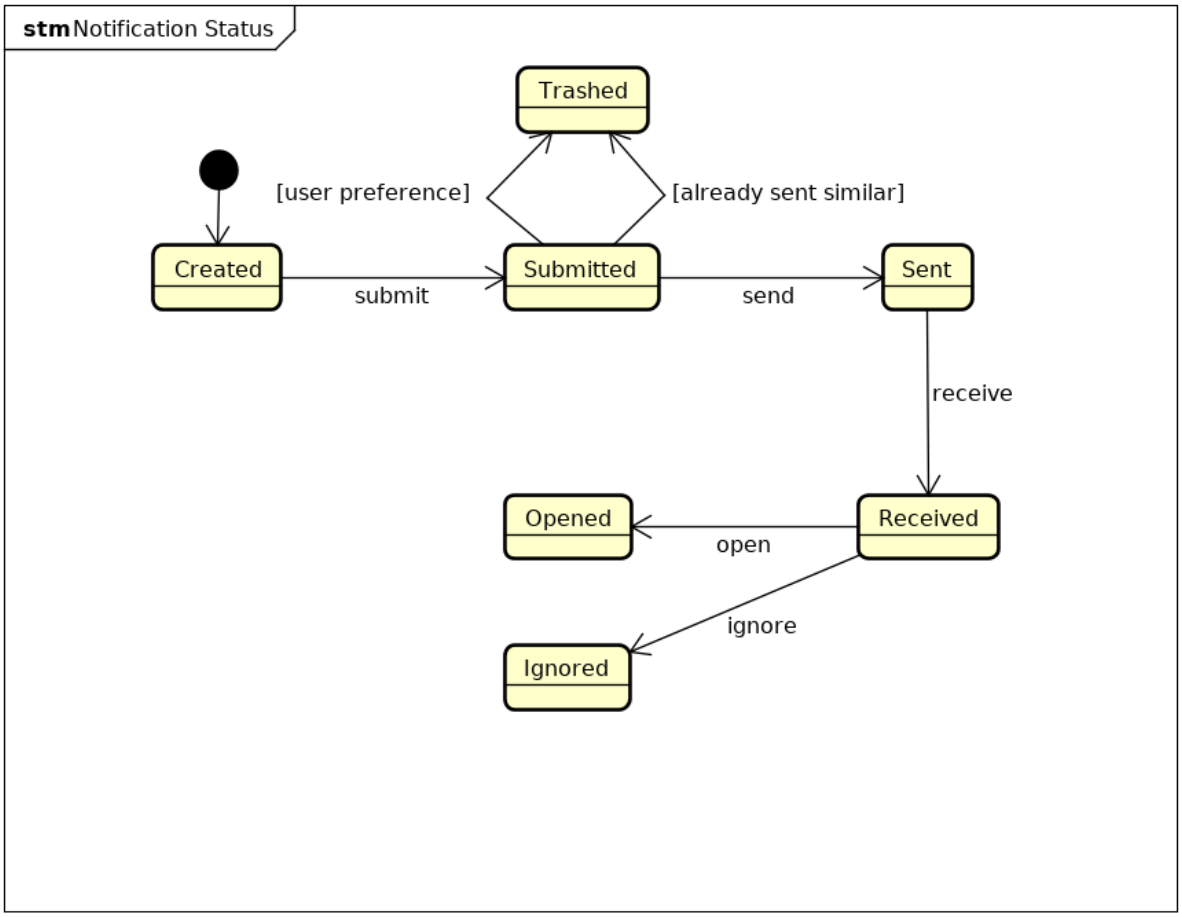

act Send manual notification
Ask for notification details
Check notification details
[invalid]
[valid]
Get User
Send Notification
[has more users]
[!users]

4.3 State Diagrams

4.3.1 Status Of A Trip

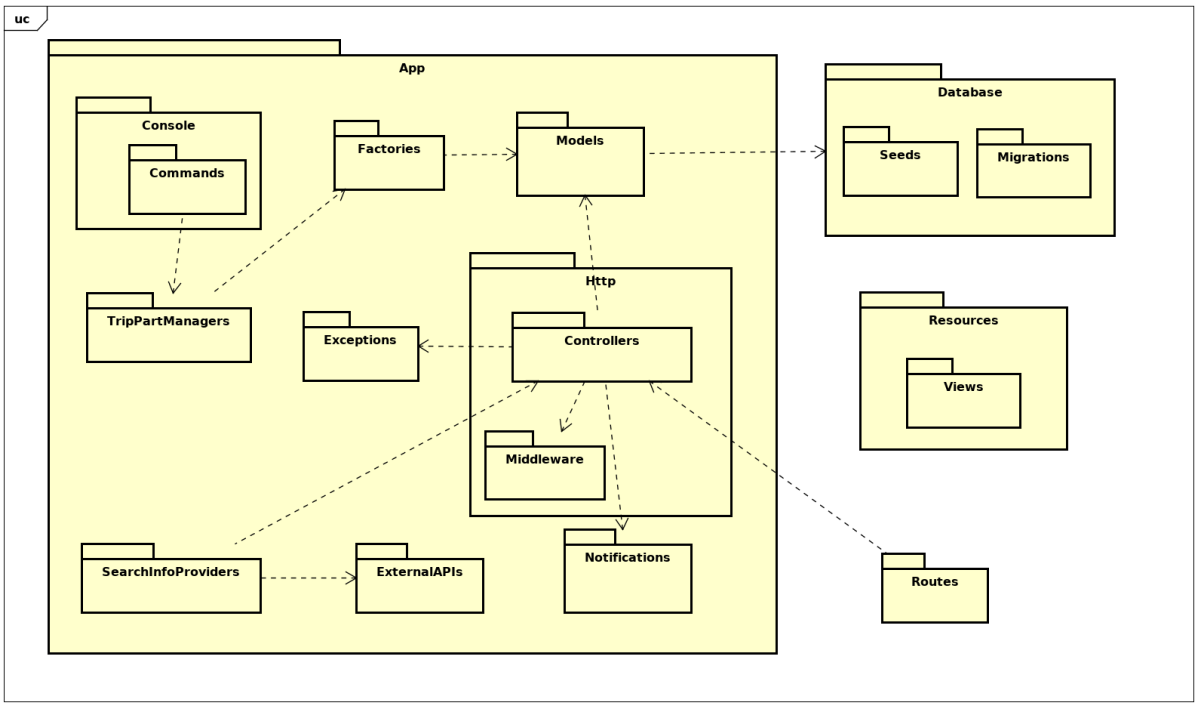


4.3.2 Notification status



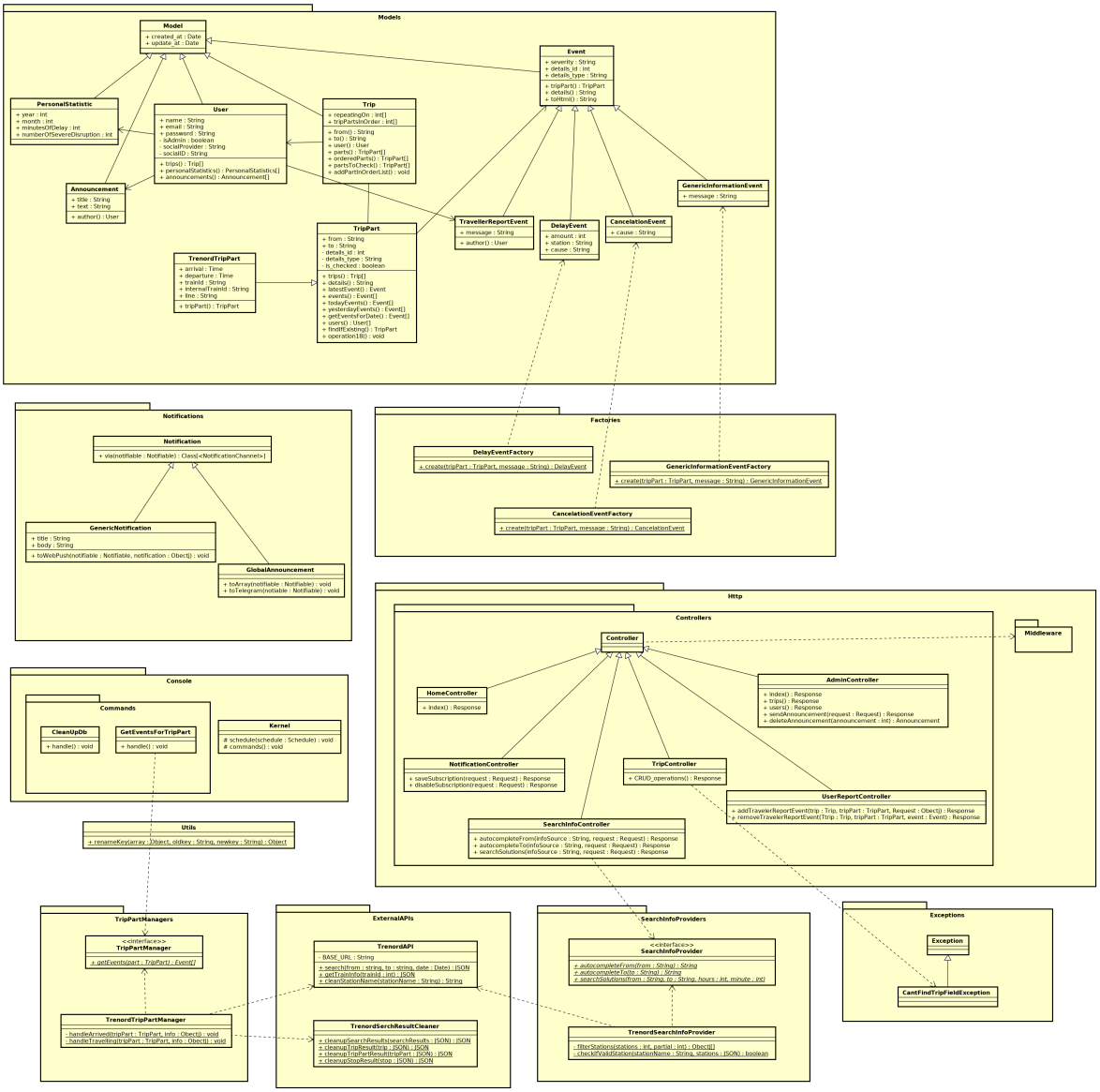
5 Development

5.1 Packages Diagram

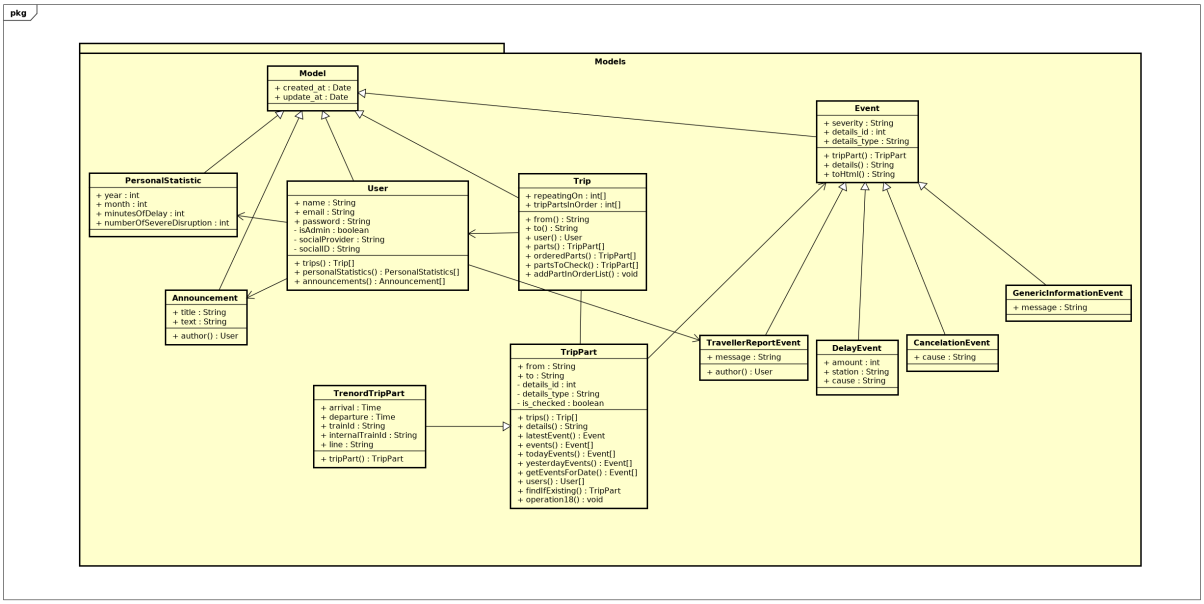


5.2 Class Diagram

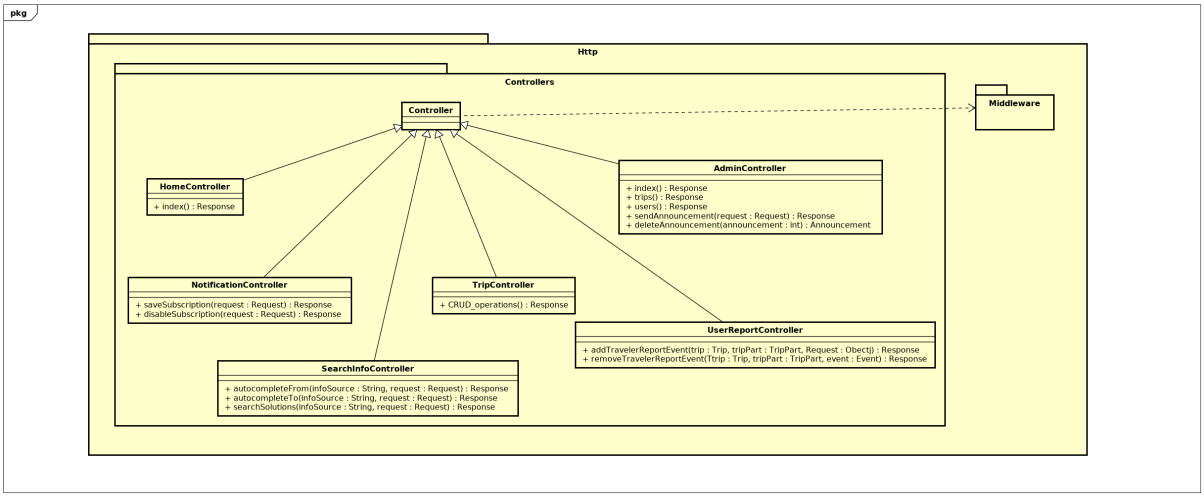
5.2.1 Overview



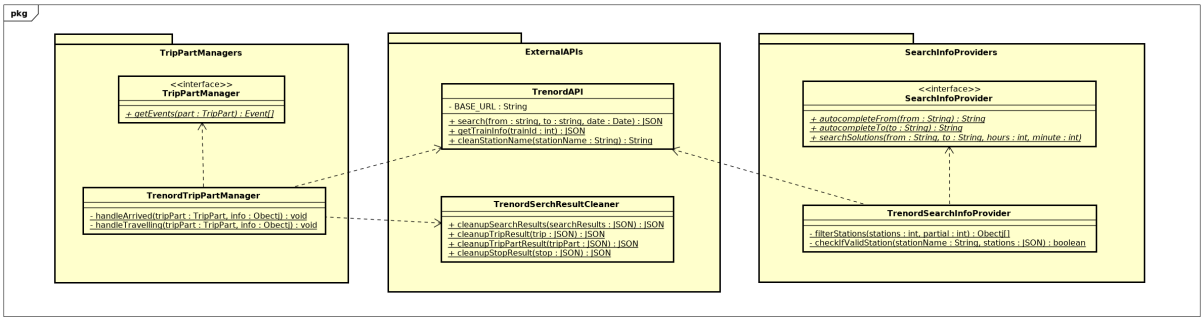
5.2.2 Detail: Models



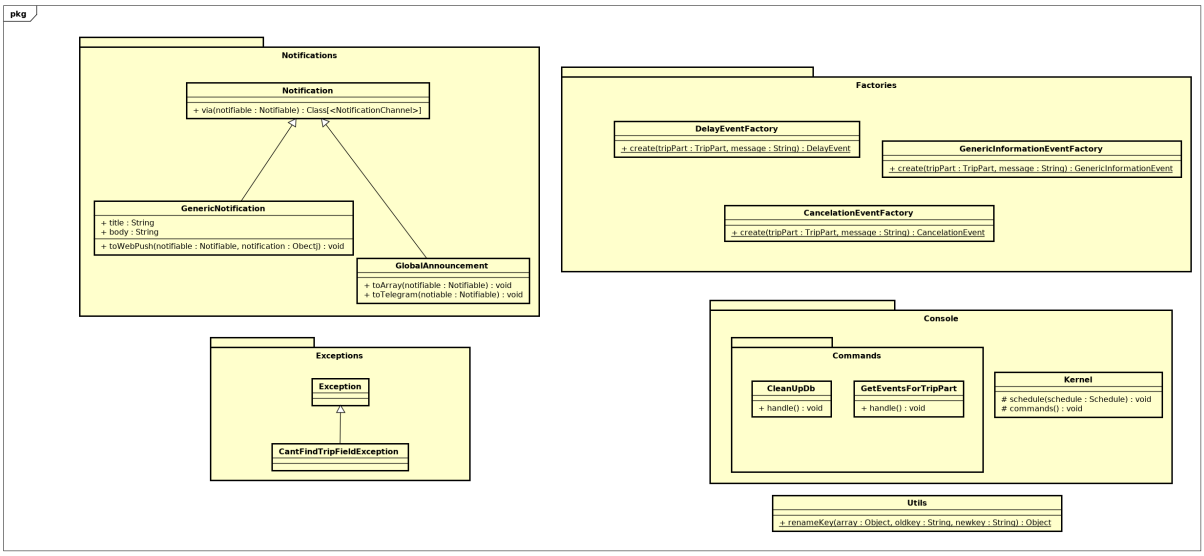
5.2.3 Detail: Controllers



5.2.4 Detail: Information Providers



5.2.5 Detail: Other



5.3 Language and Tools

To develop a Web Application for the project we used **PHP 7.1** for the backend and **HTML5, CSS3, Javascript** for the frontend.

Together with PHP we used Laravel, a framework with ease of use, cleanliness and best practices in mind. Since it came with many out-of-the box features it allowed us to focus on the design and implementation of the core business logic of the project. We enriched Laravel with a few packages to again, improve the development experience, mainly Carbon (for working with dates) and Guzzle for working with external HTTP APIs.

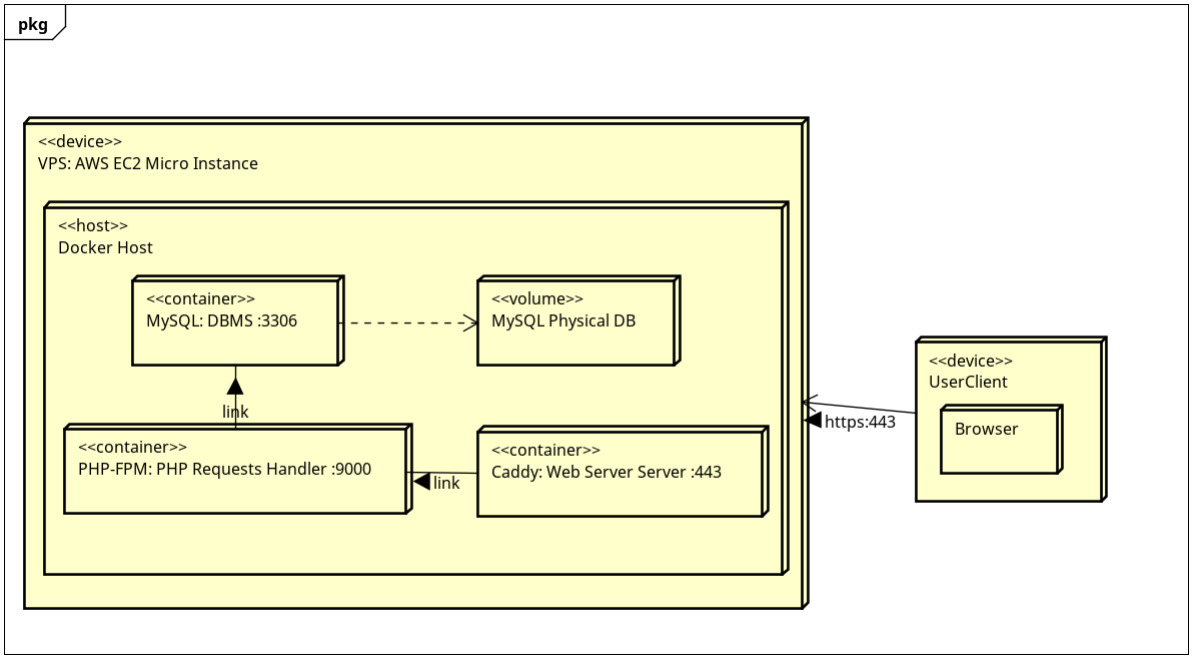
For the frontend since it was very time consuming writing the style from scratch we used Bulma.io a modern CSS framework that does provide already-made components for implementing web interfaces. We also used jQuery for handling user interaction in Javascript and modern Web APIs for sending push notifications to users (namely the Service Worker API)

For handling the best compatibility across the development and the production environment we used Docker, a container-based solution that allowed us to develop without worrying about having the same packages/operating systems. We used an existing Docker configuration, called Laradock

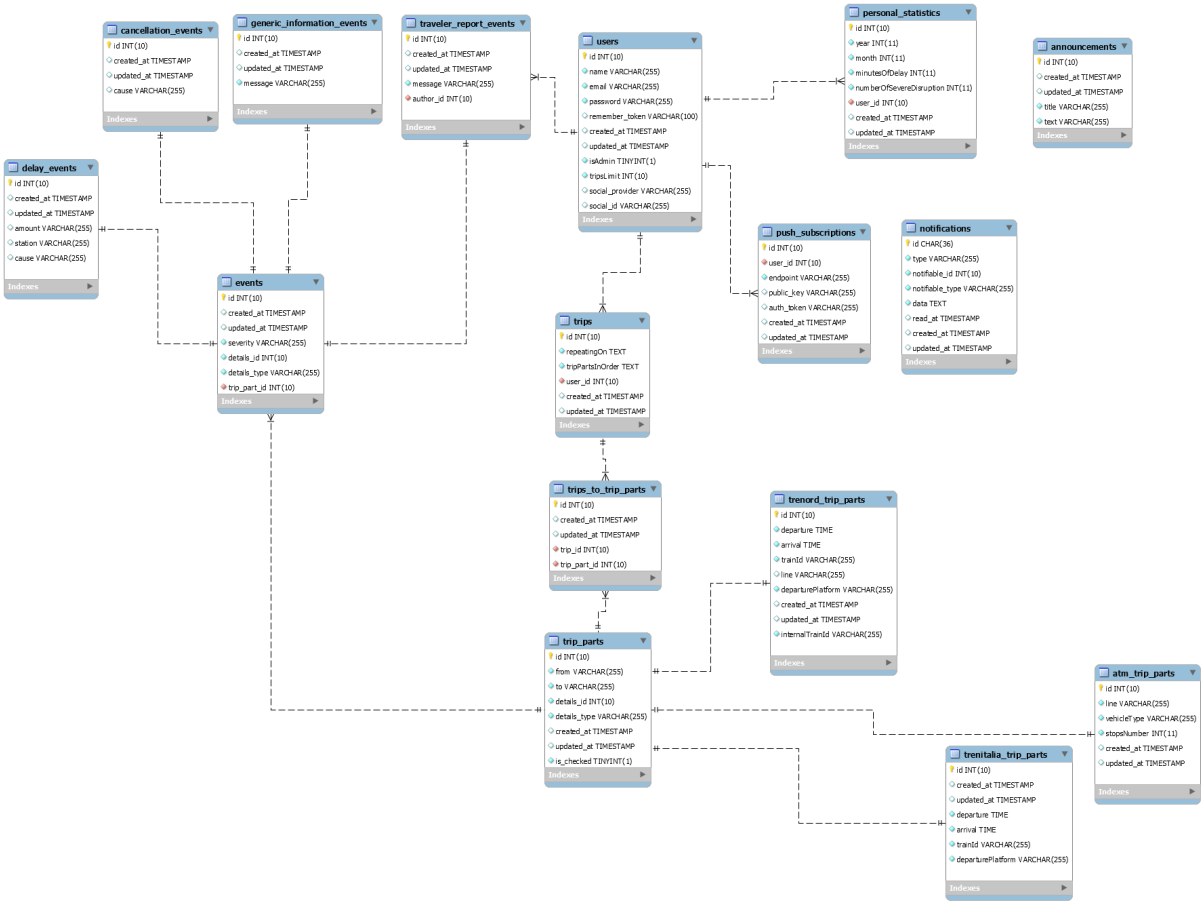
We kept track of the development using git and hosted the project on GitHub as a public repository since we decided that there was no point in hiding the code and it could be useful to others developing for the same problematic. Mostly the development took place on the `master` branch but sometiems we created feature branches when we were making braking changes and did not want to distrupt other developers. Examples of branches are the `push-notifications` branch and `improve-UI` branch.

We decided to keep track of the work to be done using a Kanban board 3-column approach, keeping track using the Projects feature of GitHub. You can see the Kanban board here.

5.4 Deployment Diagram



5.5 Database Schema



5.6 Project structure

The project is organized as a standard Laravel application, with a few custom packages and namespaces defined for our use case:

- `app/`, the code at the core of the project
  - `Console/`, custom Laravel commands for the periodic background checks
  - `Exceptions/`, custom exceptions defined for the application
  - `ExternalAPIs/`, classes that interface with external services, such as Trenord, Trenitalia and ATM APIs
  - `Factories/`, factories to build complex objects
  - `Http/`
    - `Controllers/`, the MVC controllers
      - `Auth/`, special controllers to handle user authentication

        * `Middleware/`, requests middlewares to handle connection to routes
    - `Models/`, the MVC models to handle all the objects relevant to our project
    - `Notifications/`, special objects to handle the notifications across the application and to the users
    - `Providers`, Laravel services that we did not use
    - `SearchInfoProviders`, classes that offer methods to implement search for all the various sources of informations
    - `TripPartManagers`, classes to handle the different kind of `TripPart`s and get updates on them
    - `Utils.php`, a few helper methods
- `config/`, project configuration files
- `database/`
    - `migrations/`, files describing how to build the database schema
    - `seeds/`, seeder classes to insert placeholder data into the Database after building it
- `public/`, for files to be served as-is by the Web Server
- `resources/`
    - `views/`, files needed by Laravel at run time to build and serve views to the user
- `routes/`, files defining routes (URLs) for the application
- `storage/`
    - `app/`, files that needs to be accessed by the application at runtime
- `.editorconfig`, EditorConfig file for keeping the code style consistent across developers
- `composer.json`, file containing informations on all the external packages required by the application
- `readme.md`, developer documentation for the project
- `sonar-project.properties`, SonarQube configuration file

All the other files or directories are required by Laravel and were untouched.

5.7 Patterns

Since, according to the requirement analysis, we had to develop a Web Application we decided to use an **MVC (Model View Controller)** approach for the main parts of our project. Some other parts, mainly the one that takes care of periodically checking if there are delays/events for the registered trip was developed similar to a **Broker/Worker** structure but it is a bit simpler, implemented, for example, without queues or complex services to hanle the tasks.

Since we developed the project using the PHP framework Laravel, it was quite straight forward to develop following the MVC pattern, since Models, Views and Controllers are at the base of the framework.

Another important pattern we applied for allowing a more agile workflow is the **Evolutionary Database Design**. It is well explained by Martin Fowler in this webpage. This pattern allows for faster prototyping and for keeping track of the DB Schema changes in the git repo.

In the design pattern compartment we used a few, mostly the one enforced by the framework by design and a few other because we felt it were necessary.

- **ORM Pattern/Active record**, this pattern is widely used in the project for accessing the database and for mapping the Models to the database. Provided by Laravel.
- **Builder Pattern**, used to create the queries to be executed by the ORM. Provided by Laravel.
- **Facade**, used for implementing the different data sources so that they expose a common interface. It was used together with the **Template Class** pattern.
- **Publish/Subscribe**, for handling notifications and application-wide communication.
- **Transform view**, for providing different representations of widely used objects, such as the various kinds of `Events`.
- **Factory**, used for creating complex objects such as `Events` and its specialized subobjects (`DelayEvent`, `GenericInformationEvent` and similar).

5.8 Testing the project locally

In the code repository we included some instructions on how to build and test the software locally. There are a few requirements: Docker, Docker Compose, git.

Since it is a pretty long process to build it locally (there are a few steps to follow and it can take some times to download all the packages and docker images) we decided to deploy the project here for you to try out. We also created a demo account (`demo@demo.com` with password `password`) which has admin permissions and a few trips already planned.

Warning: To also deploy a fully working version of the project (not only building and starting but also allowing the user to login via Social Networks, search and follow trips) you need a few values to insert into the `.env` files, API keys and other *secrets* that we can't (and should not) share on a version controlled public repository. If you want these values please get in touch with the group supervisor (s.vitali@campus.unimib.it)

5.9 Static Analysis with SonarQube

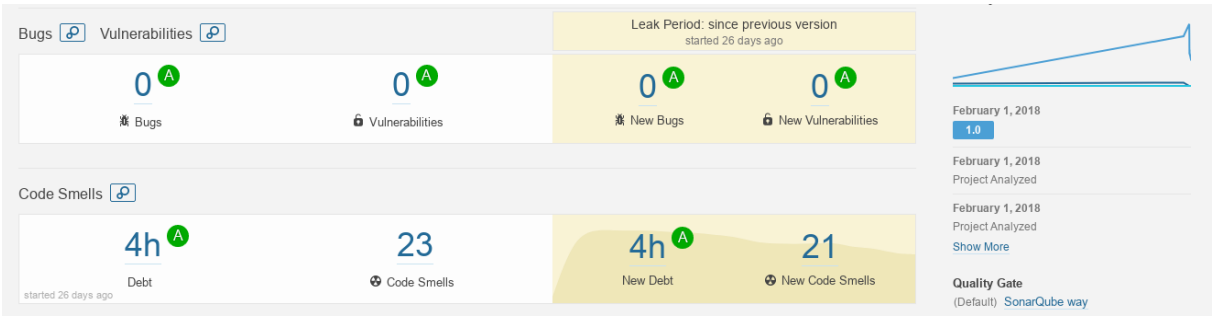
To keep track of code quality we used SonarQube together with SonarQube Scanner. This proven quite a useful tools, since it allowed us to keep track of the code as it grew and improve it as

necessary. We aimed (and got) 0 bugs and `A` level for code smells; there are still some code smells left in the code (about 20, *all* about string literals not being represented as constants) especially related to a particular class: `App\ExternalAPIs\TrenordSearchResultsCleaner`.

It is possible to fix them just to please SonarQube, but it would be *wrong* since that class is responsible for walking through an object with many different fields and removing the unnecessary ones: having constants instead of strings would not remove the complexity of the code. The same goes for the same kind of code smells in the other classes.

Another thing we did not like about the SonarQube code alayzer is that it did not "like" functions with more than 3 returns. We strongly believe in the return ealy pattern for writing functions. To avoid a lot of returns we had to use long conditions in `if` statements which we think is not ideal.

Also SonarQube did not find errors about missing includes and undefined variables. In the end we also resorted to other static analysis tools such as PHPStan that proved more effective in some fields where SonarQube was not so great.



To make SonarQube work with the project we had to create a `sonar-project.properties` file that specifies in which directories SonarQube had to look for code. After starting SonarQube and running `sonar-scanner` inside the project folder the code gets analized and the results are served in the Web Inteface of SonarQube.

5.10 Adding new information providers

The system was designed and developed to be as extensible as possible. It is possible to add as many Information Providers (different transportation services) as necessary, given that you know how to extract informations from the.

5.10.1 General Steps

- Analyze the service and the information provided

- Create a Database migration for the provider's model istances
- Create Models
- Create a SearchProvider
- Update the TripPartController
- Associate the dedicated SearchProvider class into the generic SearchController
- Create external API for retrieving data
- Display the results into dedicated views

5.10.2 Analyze the service and the information that could be used by PTM

Since the project should help travelers that use transportation vehicles you must assure that you can always identify for each new information provider:

- A departing point
- An arrival point
- A generic time around which the service will be provided

Inserted that, a list of a solutions will be researched and displayed to the users, via the `searchSolutions` method.

Tipically, the first two pieces of information are known locations suchs as public stations; in this case it should be possible to retrieve a collection of those locations to help the user selecting them.

5.10.3 Create a DB migration for the provider's model istances

You need to create a migration in `app/database/migrations` via the `php artisan make:migration` command.

A table will be added to the DB in which all the additional information identified at the previous step will be recorded.

5.10.4 Create a Model class

Create `/app/Models/TripParts/<SpecificProvider>TripPart.php`

Example: `/app/Models/TripParts/AtmTripPart.php`

Every new specialized TripPart will be represented as a specific class in the application. Each of those classes must respect the polymorhipc relationthip with the abstract class `TripPart` and to do so they must include the function:

```
public function tripPart()
{
    return $this->morphOne('App\TripPart', 'details');
}
```

5.10.5 Create a specific SearchProvider

You must create a SearchProvider class, `app/SearchProviders/<SpecificProvider>SearchInfoProvider.php`

Example: `app/SearchProviders/TrenordSearchInfoProvider.php`

This class will provide 3 methods, used to handle search requests from the user:

- `autoCompleteFrom`
- `autoCompleteTo`
- `searchSolutions`

5.10.6 Associate the dedicated SearchProvider class into the SearchController

In the class `SearchInfoController` you must add to the dictionary `infoSources` the class that will be responsible for the search, together with a label to be used when routing informations to that source.

5.10.7 Add a dedicated handler inside TripPart controller

This function will handle insertion operations on the specific trip part offered by the provider.

5.10.8 Display the results into dedicated views

Create `app/views/trips/<specificTripPartView>.blade.php`

Example: `app/views/trips/createTrain.blade.php`

To offer the best user experiences, some specific views will be created to display the information related to the solutions of that specif provider. Those views will be included into the layout of the applications.

6 Lesson Learned

6.1 Planning and Design Phase

- We understood why the design phase is fundamental in all projects, it allowed us to really think how the project was going to be implemented (especially regarding its structure)
- Working on a project about an environment you know quite closely (public transport services) definitely proven to find critical and focal aspects of the system that needed more attention than others
- Following closely the SCORE Proposal for the project proven quite a challenge because it was very wide-scoped and had a lot of features that needed to be implemented
- It was not easy to understand how much the use cases (and diagrams in general) needed to be detailed
- It is fundamental to learn how to use modelling tools efficiently to save time and improve quality of the work

6.2 Development phase

- We understimated development times during the initial design phase (also because we did not really plan for the exams in the first place)
- For some features of the backend it would have been better to develop following a Test Driven Development approach, we did not do so because we found writing tests very time consuming. In the end we think we should have written the test, since a lot of time was spent debugging
- Version control is fundamental and very useful, given that you know how to use it; it sped up the development allowing us to parallelize the work in all the different parts of the project
- Keeping track of tasks that needed to be done was very helpful because it allowed us to focus on work that had the highest priority
- Developing both the backend and the frontend at the same time is neither easy or fast, especially if there are involved different technologies and frameworks

  - If we could have divided ourselves between working on just the backend or on just the frontend, it would have been much faster and easier for us but we could not have learned about all the different aspects of developing a web application
- Using Docker for the development and production environment was a very good choice, we focused on the code and let the containers do the deployment and the machine setup
- Even if we mostly knew PHP working with a framework was very different than just writing simple PHP Code, some time had to be spent on training and learning the framework
  - In the end using a framework was the best choice since the resulting code was much cleaner, easier to change and less bug-prone
- SonarQube is useful, but is not enough. The PHP integration has not everything we wanted and in the end we had to use other tools for static analysis (such as PHPStan)

6.2.1 Technical Difficulties with extenal APIs

- Working with external APIs, especially APIs not publicly available (Trenord, Trenitalia, ...) was not ideal, we spent a lot of time figuring out how the APIs worked, since they were not documented
  - If we had developed the service with a different country in mind it would have been a bit easier, since most foreign public transports offer a documented and publicly accessible API

6.3 Other considerations

6.3.1 Quality

- If we had more time we could have completed all the features specified both in the original proposal and in the planning phase, as well as extensive QA testing and and improved user interface

6.3.2 Risks

- External APIs are not reliable, a few times they stopped working and we had to figure out how to handle these failures
  - There was and there is still the concrete risk that we could get blocked from the informations providers