T-404-LOKA, Final Project, 2016-1

Færð

App for Road Conditions

Dovydas Stankevicius
Gunnhildur Finnsdóttir
Telma Guðbjörg Eyþórsdóttir

Contents

1. **Introduction** **1**
   - 1.1 Færð . . . 1
   - 1.2 Android prototype . . . 1
   - 1.3 Cross-platform app . . . 2

2. **Project Organization** **4**
   - 2.1 Scrum . . . 4
   - 2.2 Sprints . . . 4
   - 2.3 Communication . . . 6
   - 2.4 Report Schedule . . . 6

3. **Release Planning** **7**
   - 3.1 User Roles . . . 7
   - 3.2 Product Backlog . . . 8
   - 3.3 Initial Velocity . . . 10

4. **Risk Analysis** **11**
   - 4.1 Development . . . 11
   - 4.2 Production. . . . 12

5. **Design** **13**
   - 5.1 System Overview . . . 13
   - 5.2 Backend Design . . . 13
   - 5.3 User interface Design. . . . 15

6. **Quality Assurance** **19**
   - 6.1 Programming Disciplines . . . 19
   - 6.2 Documentation . . . 19
   - 6.3 Deployment time . . . 19
     - 6.3.1 Faerdapi . . . 19
     - 6.3.2 Faerd-client . . . 19
   - 6.4 Continuous Deployment – server . . . 20
   - 6.5 Continuous Integration – client . . . 20
   - 6.6 Security . . . 21

7. **Testing** **23**
   - 7.1 Unit Testing . . . 23
     - 7.1.1 Client . . . 23
     - 7.1.2 Server . . . 23
   - 7.2 Acceptance Testing – server . . . 24
   - 7.3 Load Testing – server . . . 24
   - 7.4 Known Bugs . . . 24

8. **Progress** **25**
   - 8.1 Sprint 1 . . . 25
   - 8.2 Sprint 2 . . . 26
   - 8.3 Sprint 3 . . . 28
   - 8.4 Sprint 4 . . . 29
   - 8.5 Sprint 5 . . . 31

8.6 Sprint 6 32
8.7 Sprint 7 33
8.8 Sprint 8 34
8.9 Sprint 9 35
8.10 Sprint Summary 37
8.10.1 Time spent 37
8.10.2 Product Backlog 38

9. **Operations Manual** **40**
9.1 Client 40
9.2 Server 41
9.2.1 Development machines 41
9.2.2 Build server 42

10. **User Manual** **47**
10.1 Downloading the app 47
10.2 Using the app. 47

11. **Conclusions** **54**
11.1 Results 54
11.2 Lessons learned 54
11.3 Next steps 55

1. Introduction

1.1 Færð

Færð is a mobile app where travelers on Icelandic roads can access information on road conditions and other useful data regarding their trip. It consists of two main components, a mobile client with a simple user interface and a back-end where data is collected, merged and parsed. It is designed with both Icelandic locals and an ever increasing number of foreign visitors in mind. Færð was created by three students at Reykjavík University with support from The Icelandic Road and Coastal Administration (Vegagerðin).

Vegagerðin collects many different sets of data regarding the transport infrastructure of Iceland and its condition at any given time. This information is updated very frequently and includes for example data from weather stations, pictures from web cameras, the location of roads and sections currently under construction. Although this information is very well maintained and partially available through an API, it has not been presented to the public in an accessible manner, especially considering that the majority of internet traffic is now generated by mobile devices such as smartphones and tablets.

Vegagerðin does have a website where the most relevant information for drivers is displayed, however its usability is compromised on mobile devices due to the fact that the display is overcrowded with data and presented in static maps that the user has to select one after the other on longer routes.

The purpose of Færð is to improve access to useful information for drivers through mobile devices. To reach that goal the available information has to be carefully considered and prioritized based on how relevant it is to the user, it has to be fetched and processed and finally displayed and made accessible in a user friendly and intuitive interface. In Færð the data from Vegagerðin is augmented with other information such as routing path and gas prices.

Figure 1: *Vegagerdin.is on iPhone 5*

1.2 Prototype for Android

The first iteration of this project took place in a six week long Advanced Android Programming course offered by the computer science department of Reykjavík University in the fall semester in 2015 where Gunnhildur Finnsdóttir and Dovydas Stankevicius created a simple but fully functioning prototype of the app written in Java and built according to client-server architecture. The client was organized into three Android activities, the start screen where user selected location and destination, a simple splash screen that appeared while the data was fetched and an activity to display the results that was divided into three fragments, one with an estimate of time and cost of the trip, one with a map showing the route with color coded information about road conditions and finally one where the user could click through a selection of photos from web cameras along the route. The primary

purpose of the server was to combine and extract data from several data sources and to return relative road conditions, webcams and other relevant information to the app user. The server was written using NodeJS and hosted on Heroku.

By building this first version of Færð, the developers became very familiar with the data and designed algorithms to combine different datasets to create useful information for the user. The general structure of the user interface with a screen for selection and results displayed on a map was also designed during this part of the project. This version was however lacking in many regards, most importantly it was only built for a single platform and the data displayed was in some edge cases not accurate because of pollution in the data. This prototype was also unreliable due to the routing service (OSRM) often being down which rendered the app useless until the service was back online.

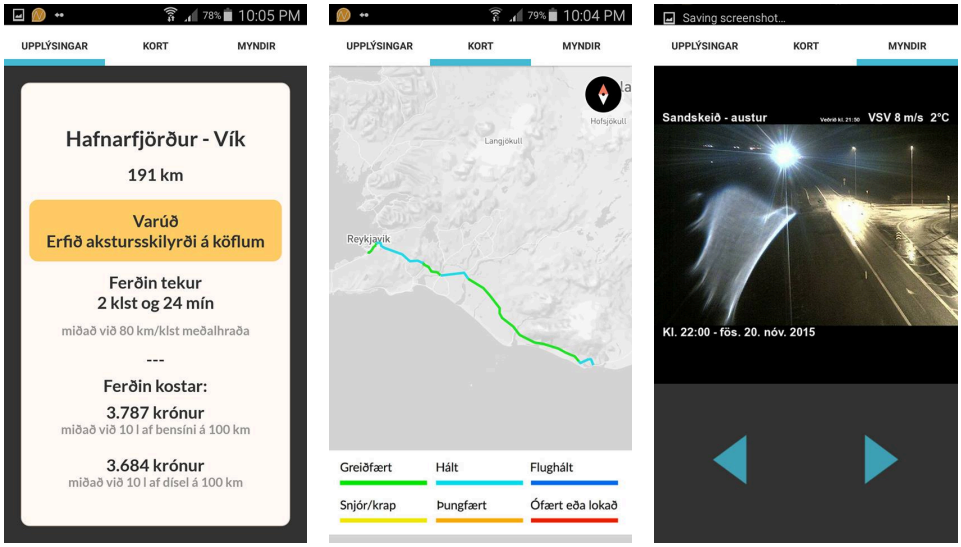


Figure 2: *Three fragments displaying results in Færð prototype*

1.3 Cross-platform app

The latest version of Færð and the subject of this report was built as a final project for a BSc degree at Reykjavík University. In this iteration the scope of the project was greatly increased because of a number of factors. A new developer, software engineering student Telma Guðbjörg Eyþórsdóttir joined the project and the timeframe for development was larger, stretching over a 15-week period. The experience gained from the first iteration gave the team a solid understanding of the challenges involved and solutions to some of the more difficult problems that needed to be solved. Lastly the team entered into formal collaboration with Vegagerðin and was supplied office space, server hosting and access to technical staff for this version of the project. Vegagerðin intends to publish the app at the end of summer 2016.

The main goals at the beginning of this development cycle were the following:

1. Creating an app for both Android and iOS. According to data from Vegagerðin, 51% of the mobile users who visit their web use iOS while 46% use Android devices and the remaining 3% divided between other operating systems. It is therefore critical to serve both these user groups when developing an application that is meant for the general public. Since the time constraints did not allow for two native apps to be built the team decided to use a JavaScript framework to build for both platforms at once.

At the start of client development, the possibility of using React Native was explored but as it is a very recently developed framework that has limited documentation, the team decided to switch to the more stable Ionic framework.

2. Introducing robust unit testing on both client and server to ensure the stability and reliability of the app. The goal was to have 70% total code coverage with all business logic thoroughly tested.
3. Load balanced servers. Since this version was developed with the goal of publishing in mind, the team had to design the server to ensure that the it would handle heavy traffic and be easily scalable. The initial setup was planned to include a Nginx load balancer and two Node clusters.
4. Continuous deployment on the server and continuous integration on the client. The source code is version controlled on GitHub and all changes to the server code pushed to the repository are automatically deployed using a Jenkins build server through development, testing and production stages in virtual environments. On the client side, the application can not be fully deployed automatically, but a similar system for automated tests will be set up.
5. Added client features. The design for this iteration included many features that were not a part of the prototype but greatly enhance the usability of the app. Examples include displaying weather data, supporting a selection of languages and making the map where results are displayed interactive with more information available on tap. A full list of features can be found in the product backlog later in this report.
6. As this was a full-scale software development project, it was decided to use the tools provided by the Scrum methodology. This included organizing the work on the project into sprints and constantly updating documentation throughout the period.

The team believes that all these goals have been achieved and that the application meets the requirements laid out in the beginning of the project. The project has progressed evenly and steadily throughout the time period despite the team having to adjust to some challenging situations along the way. The end result is useful software that is robust and ready for publishing.

2. Project Organization

2.1 Scrum

Our group consist of three students at Reykjavik University and product owner from Vegagerðin. The group was divided into three major roles: Scrum master, Product Owner and Development team. Each role is very important for the project development. For example, product owner has responsibilities, such as deciding on release dates and content, prioritize features according to market value or even accept or reject work results. While in contrast, the Scrum master's responsibility is to represent management to the project, hence responsible for enacting Scrum values and practices. Table below summarizes the roles and the team members assigned for each role:

Team roles	
Role	**Team member**
Product owner	Viktor Steinarsson
Scrum master	Telma Guðbjörg Eyþórsdóttir
The team	Dovydas Stankevicius, Gunnhildur Finnsdóttir, Telma Guðbjörg Eyþórsdóttir

Table 1: *Role summary*

The product owner for this project was Viktor Steinarsson, the head of IT at The Icelandic Road and Coastal Administration. He is an excellent candidate for being the product owner as he has plenty of experience in the field. We declared Telma Guðbjörg Eyþórsdóttir to be the Scrum master, because she has an excellent insight on how scrum processes work and is very good in supervising and enhancing the team spirit. Finally, our team was fairly small, therefore all of the team members were a part of the development team in addition to their roles.

2.2 Sprints

As we did this project along with other school work, for the first 12 weeks of the project we had two-week sprints as we could not dedicate our full attention to the project. More specifically, we dedicated 40% of our available work time to this project (12/30 ECTS), which was approximately two work days per week. However, after completing the final exams, we reduced the sprint length to one week as we worked 100% on this project, hence full work week. Table below summarizes the sprints:

Sprints				
No.	**Start**	**End**	**Length (weeks)**	**Capacity (hours)**
0	11.01.16	31.01.16	3	180
1	01.02.16	12.02.16	2	120
2	15.02.16	26.02.16	2	120
3	29.02.16	11.03.16	2	120
4	14.03.16	25.03.16	2	120
5	28.03.16	8.04.16	2	120
6	11.04.16	22.04.16	2	120
7	25.04.16	01.05.16	1	150
8	02.05.16	08.05.16	1	150
9	09.05.16	12.05.16	1	120

Table 2: *summarizes sprint schedule and each sprint's max capacity*

We calculated the sprint capacity for two-week sprints as follows:

2 work days * 10 hours * 3 developers * 2 weeks = 120 hours

We calculated the sprint capacity for one-week sprints as follows:

5 work days * 10 hours * 3 developers * 1 week = 150 hours

During sprint zero, we spend approximately 180 combined hours in three weeks, where we used the time to plan the project, design the API, attend meetings with Vegagerðin and do research about the technology that we are going to use in this project. For example, we took an online course on React Native that contained over 8 hours of lectures because we had to learn it from scratch as we had no previous experience working with the framework. We decided not to include sprint zero when calculating total project capacity, however we did consider it when estimating the initial velocity. Finally, the last sprint did only consist of documentation and revising the code.

Before the start of each sprint we had a sprint planning meeting where the entire team met up and created the sprint backlog from the product backlog. The members of team defined tasks they could commit to complete, in other words based on the capacity of the sprint.

By the end of each sprint we had sprint retrospectives and sprint reviews. The sprint retrospectives were typically around 30 minute sessions, where entire team participated and looked at what was and wasn't working and discussed how we could improve in the upcoming sprints. We structured our retrospective meetings according to James Shore retrospective format.

In the sprint review on the other hand, entire team met at Vegagerðin and held open sprint reviews where anybody could participate. The sprint reviews were informal meetings with around two-hour preparation rule and no slides were used. During the sprint review we presented what we had accomplished during the sprint, typically in the form of a demo, where we introduced new features and underlying architecture.

Furthermore, we had arranged an office space at Vegagerðin to work on the project as a group every Monday and Thursday afternoon. The rest of the work we did individually or in pairs. Finally, our sprint backlog was shared on team Dropbox. Every time a team member

chose to start working on a certain feature, he/she had to sign their name, so other team member would be informed what had been done - in progress – was available, as well as, daily updating remaining work estimated for each feature.

2.3 Communication

As effective means of communication is a central part of Scrum, we decided to use Slack. For organizing our discussions, we created different channels for each part of the project i.e. channel for API, channel for the client, where we keep the discussion specific to the topic. Additionally, we integrated Google calendar, GitHub and Jenkins plugins to push important notifications, such as build failures, new GitHub issues, to an appropriate Slack channel where we got immediate push notifications to our smartphones immediately informing us with important updates.

To manage our code, we created a private organization on GitHub, where each part of the project (i.e. client, API) is stored in different repository. We decided to follow this model in order to create a separation of concern, so we treated each part of the project independent of another.

2.4 Report Schedule

Report schedule	
28.1. Initial documentation	Project proposal Project organization report Product backlog draft
08.2. First progress report:	Revised project proposal Revised project organization report Release Planning Risk analysis Design draft Progress report
11.3: Second progress report:	Sprint log
10.5: Third progress report:	Operations manual User manual Design report
13.5: Final hand in:	Revised operations manual Revised user manual Revised design report

Table 3: *Report Schedule*

3. Release planning

3.1 User Roles

One of the most important aspects in designing successful software is understanding its user. After performing some analysis, we divided the users into the following groups and summarized their backgrounds, use of system and context of use.

User groups				
User Group	Background	Use of the system	Context	Main Goals
Users	**Age**: 17+ **Gender:** All **Education:** Varies **Disabilities:** Varies **Comp Skills:** General	Seeing relevant data for road safety	On the road with mobile devices	To have a good idea about the road conditions, weather, and other data for a chosen trip.
Tourists	**Age**: 21+ **Gender:** All **Education:** Varies **Disabilities:** Varies **Comp Skills:** General	For driving in Iceland and seeing data for road safety/routing	On the road with mobile devices	To have a good idea about the road conditions, weather, and other data for a chosen trip.
Locals	**Age**: 17+ **Gender:** All **Education:** Varies **Disabilities:** Varies **Comp Skills:** General	Seeing relevant data for road safety	On the road with mobile devices	To have a good idea about the road conditions, weather, and other data for a chosen trip.
Drivers in bigger vehicles	**Age**: 25+ **Gender:** All **Education:** Varies **Disabilities:** Varies **Comp Skills:** General	Seeing relevant data for road safety, mainly wind gusts	On the road with mobile devices	To have a good idea about the road conditions, weather, and other data for a chosen trip.
Developers	**Age**: 22+ **Gender:** All **Education:** A background or education in computer science. **Disabilities:** Varies **Comp Skills:** High	Developing, testing and deploying	In the office	Create a user friendly app to check on road conditions, weather, and other data for a chosen trip.

System Administrator	**Age**: 22+ **Gender:** All **Education:** A background or education in computer science. **Disabilities:** Varies **Comp Skills:** High	Preventing from spamming, monitoring traffic, monitoring the content, fixing glitches, communicating with developers.	In the office	To add/modify data to the app and to maintain it.

Table 4: *User groups*

3.2 Product backlog

The product backlog in Scrum is a prioritized list, containing short descriptions of the functionality of the product. The Scrum team, along with our product owner from Vegagerðin, met and discussed the requirements of the app as well as some user stories for the product backlog. The team ordered the user stories into rankings, prioritizing ranking from A to C, where a story ranked A is essential to the project, stories ranked B will be included but are not as important while C ranked stories are nice to have. Afterwards the team assigned story points to each user stories using the poker planning method. Here are the following requirements from our product backlog.

A requirements	
1	As a developer, I want to have a clear overview over the project (planning)
2	As an app user, I want the app to auto-detect my current location
3	As an app user, I want to manually specify from/to locations within Iceland
4	As an app user, I want to see a total distance of my route
5	As an app user, I want to see weather data that is relevant to my route
6	As an app user, I want to see relevant road conditions color-keyed on the map
7	As an app user, I want to see all relevant webcams, presented on the map
8	As an app user, I want to be able to run the app on an Android device, compatible with API 19+
9	As an app user, I want to be able to run the app on a iOS device
10	As an app user, I want to be able switch between English and Icelandic languages
11	As an app user, I want to be able to switch between US and EU unit systems (liters vs. gallons, kilometers vs miles, Celsius vs Fahrenheit)
12	As an app user, I want to be able to see alternative routes to my destination
13	As an app user, I want to be able to see warning signs on the map if there are high wind-gusts on my route
14	As an app user, I want to be able to control/filter the amount of information that is available for my route
15	As an app user, I want to be able to see when conditions were last updated
16	As a developer, I want to be able to see API documentation
17	As a developer, I want to be able to automatically deploy my latest changes by pushing code on master branch
18	As a developer, I want to develop the software in environment as similar as possible to the production environment (vagrant)

Table 5: *A requirements*

B requirements	
19	As an app user, I want to select two locations from the map
20	As an app user, I want to be able to see latest warnings/notifications/important news from Vegagerðin
21	As an app user, I want to get a warning if my route contains any weight restrictions
22	As an app user, I want to be able to see the road conditions/weather data/web cams for entire Iceland without specifying a route
23	As an app user, I want to see estimate time of my journey
24	As an app user, I want to be able to send picture/comments to the Vegagerðin regarding road damage or other relevant information
25	As an app user, I want to be able to see road temperature
26	As a system administrator, I want to be able to see warnings/comments submitted by the app users
27	As a system administrator, I want to be able to log into an app admin page
28	As a system administrator, I want to be able to add/delete/modify locations that are available to the app users
29	As a system administrator, I want to be able to see traffic on the server in real time
30	As a developer, I want to get automatic push notification on slack in case of any new issues opened on GitHub, any changes to the team calendar, build failures to an appropriate slack channel
31	As a developer, I want to be able to deploy new node server to a new production environment by executing a single script
32	As a developer, I want the production code to have an approximately 70% code coverage
33	As a developer, I want to have acceptance test for every single API end-point
34	As a developer, I want to be able to migrate the database
35	As a system administrator, I want to be able to issue a https request to the api to update the database

Table 6: *B requirements*

C requirements	
36	As an app user, I want to see if there are any traffic jams on my route
37	As an app user, I want to be able to specify my vehicle plate number and get better estimation of my travel cost
38	As an app user, I want to be able to use the app in more than 2 languages
39	As an app user, I want to get push notification if I'm approaching extreme road conditions, such as sand storm, extreme wind, closed road, etc.
40	As a system administrator, I want to be able to see app user statistics, logs
41	As a system administrator, I want to be able to see API documentation on the admin page
42	As a color blind user, I want to be able to see road conditions drawn on the map, alternatively to color keys
43	As a developer, I want to be able to backup the database

Table 7: *C requirements*

A more detailed product backlog is provided in the attached ProgressReport.xlsx file along with each sprint planning.

3.3 Initial Velocity

The initial velocity was an estimate of how many story points we would be able to complete during our first sprint. Upon completion of more sprints, the team adjusted the velocity value as more data was obtained. Our product backlog had a total of 263 story points. To get a good estimation, we took a story with 13 story points and divided it into small tasks which took a combined time of 32 hours to complete it. By that we could conclude that each story point takes around 3 hours. According to our estimation of sprint one, the team sprint capacity was 120 hours. From this data, we calculated our initial velocity by dividing the sprint capacity by number of hours it takes to complete a story point.

Initial velocity: 120 / 3 = 40

The team velocity was 40 and the first six sprints were two weeks long. According to our initial velocity calculations we estimated to be able to complete 40 story points during our first sprint. Given we had nine sprints, where the last three sprints will be one week long, we planned to accomplish around 360 story points. Which meant we could meet the given target with room for error and risks.

4. Risk Analysis

The following is an analysis of the potential risks that the project involves. Each risk is numbered and an estimate is given of the odds of that particular issue arising on a scale of 1 to 5. The effects in terms of delay of development or user experience are also estimated on the same numerical scale. The priority of each risk is then calculated by multiplying the previous two numbers which gives an idea of the importance of minimizing that risk. The risks are mapped to appropriate actions that would minimize or counterbalance their effects and finally each risk is assigned to a team member that will be responsible for the action.

We divide the risks into development risks (things that can affect development) and production risks (things that can affect the function of the app when it's published).

4.1 Development

Development Risks:						
No.	**Risk**	**Odds**	**Effect**	**Priority**	**Actions**	**Assigned**
1	Sickness or family emergencies	2	3	6	Clear and constant team communication	Telma
2	Deadlines not met	1	4	4	Thorough planning of the project workload	Telma
3	Development hardware failing	2	3	6	All code and reports are kept in source control/cloud so work will not be lost	Dovydas
4	Delay setting up work environment	1	3	3	Solved February 10th.	Gunnhildur
5	Lack of experience with React Native	3	3	9	Do online tutorials before starting development on project. Use resources such as discussions on GitHub and Stack Overflow to get help.	Telma
6	Other courses taking too much time	3	3	9	Plan workload around deadlines in other courses.	Dovydas
7	Server side security breach	4	4	16	Collaborate with Vegagerðin to ensure server security	Dovydas

Table 8: *Potential development risks*

4.2 Production

Production Risks:						
No.	**Risk**	**Odds**	**Effect**	**Priority**	**Actions**	**Assigned**
8	Server going down due to traffic	2	4	8	Scale the server vertically in time	Dovydas
9	Routing API going down	2	4	8	Chose the API carefully	Gunnhildur
10	Vegagerðin API failing	1	4	4	Work closely with Vegagerðin to anticipate issues	Gunnhildur
11	App not compatible with both platforms	1	4	4	Testing each feature on multiple devices	Gunnhildur
12	Bugs in React Native since it is a new framework	4	4	16	Be well informed of the stability of the system	Telma
13	Dependencies becoming deprecated	3	3	9	Develop a solid maintenance plan	Dovydas
14	Failure of database	2	4	8	Regularly backup the database	Telma
15	Loss of data from message queue	1	3	3	Save messages to the disk	Dovydas
16	App providing inconsistent or incomplete information	1	4	4	Designing core algorithms carefully and using tests to verify them	Gunnhildur
17	Hardware failures on production machines	5	3	15	Working closely with the system administrator at Vegagerðin	Dovydas
18	Bugs in ionic	2	2	4	Create patches or make sure to update to the latest version	Telma
19	Server side security breach	4	4	16	Collaborate with Vegagerðin to ensure server security	Telma

Table 9: *Potential production risks*

5. Design

5.1 System Overview

The system has client/server architecture and is written completely from scratch. Client is written using Ionic framework and issues API calls to the server (faerdapi) for getting the latest road data. Faerdapi requests all the latest data from different APIs to form a mashup. Then the web service extracts all of the relevant data and replies it to the client. Figure below summarizes the general system architecture:

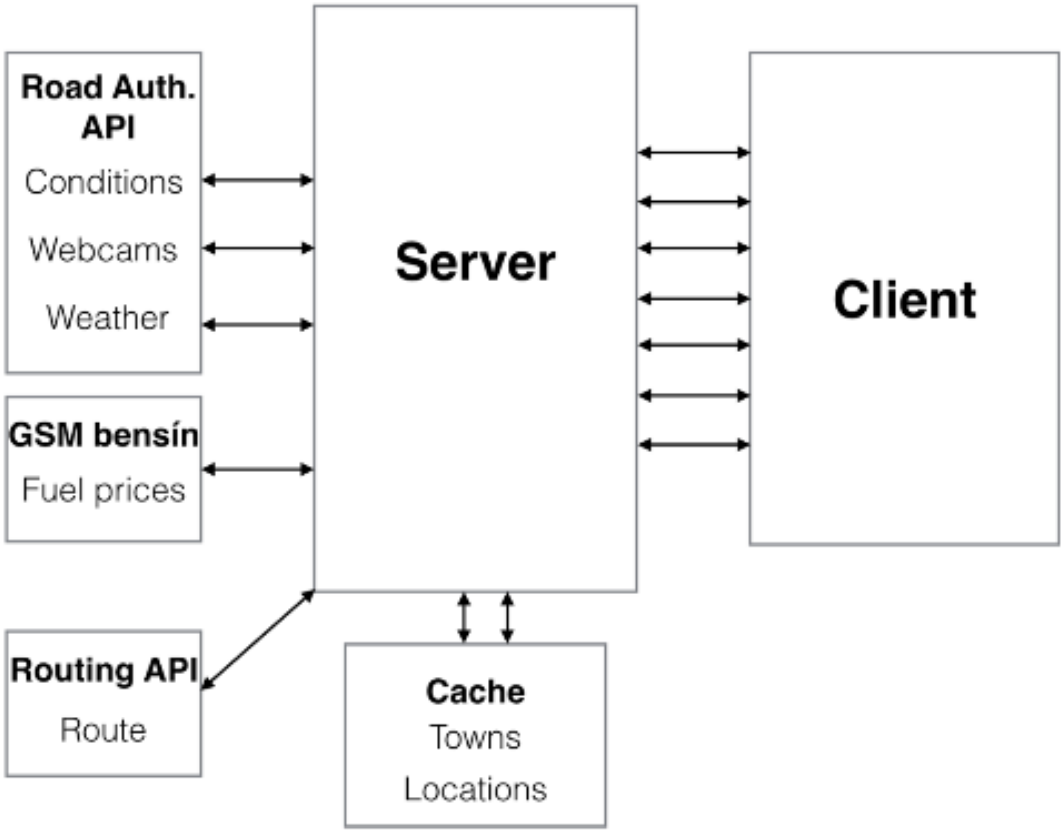


Figure 3: *General system architecture*

5.2 Backend Design

In order to utilize the system resources more efficiently, we set up a Node cluster. On startup, the Node server counts the number of available CPUs on the machine, and forks itself accordingly. This way the production server is more efficient as it uses all of the available system resources, hence is capable of handling more concurrent requests. Docker is used to wrap the web service into an Ubuntu 14.04 LTS container and abstracting the entire software from the host operating system. Using this technology, the API can be deployed to several machines running any operating system that is capable of executing Docker. Furthermore, Docker is used to track the running processes of the API and in the case of a complete shutdown, Docker process manager reboots the API, thus ultimately ensuring the continuous up time.

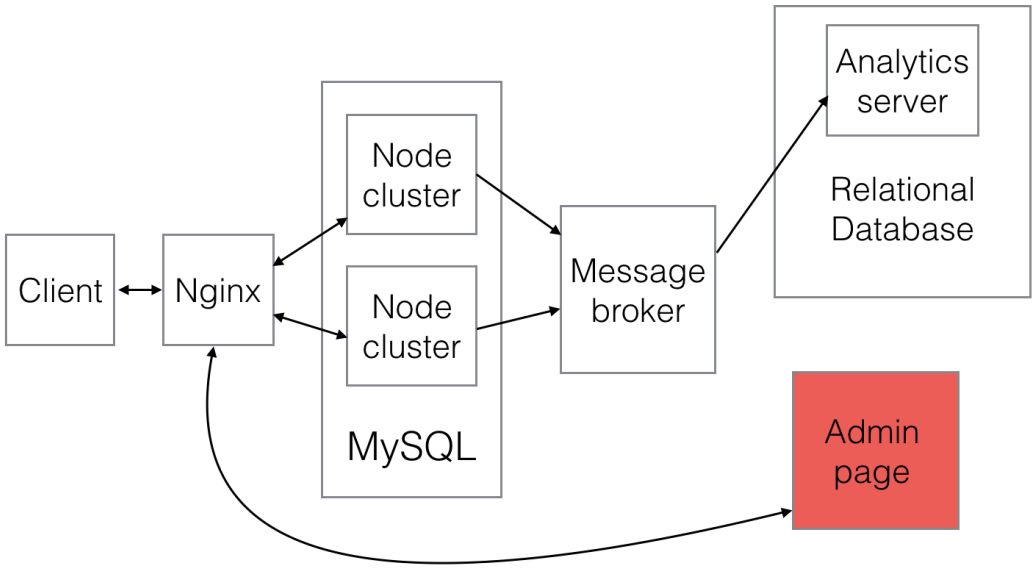


Figure 4: *Backend architecture*

Nginx load balancer is used in order to make the web service scalable and able to handle high throughput. The primary role of Nginx server is to load balance the traffic between the Node clusters and cache static files. The current architecture includes two separate production machines running an instance of our API and the traffic is balanced between them. The Nginx server provides an extra layer of protection in a case of one Node server crashing. In a case of unexpected error in one of the production servers, the API would still be available, as the Nginx would automatically forward the traffic to the running instance of the API while the other server takes time to recover.

The API has production, test and development environments. Development and test environments are configured to have exact same configuration as production environment. In order to avoid “works on my machine” syndrome Vagrant virtual box for mimicking the production environment on development machines is used. Using Vagrant allows the team members to share their exact machine configurations and provides an easy way to start from scratch if necessary. Furthermore, by using vagrant technology it is trivial to add new developers without a hassle of configuring their development machines as all configuration is stored in a virtual image.

We have implemented an analytic server which uses message broker called Node-Kafka. In the analytics server implementation, it was of primary importance to use message broker as this technology does not affect the response time of the API. All metric data is forwarded to a separate analytic server, in a form of a distributed message queue, meaning that all production servers are forwarding data to central analytic server. There data is analyzed and all the metrics are saved in the database.

Analytic server introduces several benefits to the project. App metrics can be used to improve caching on the production servers (data can be recalculated) for even faster response time. As the app displays all available Icelandic towns in form of a list, top searched towns are displayed at the top. This data representation is an excellent source for tourist to see most popular locations in Iceland. Furthermore, this is also very useful for locals, as it is not required to scroll through the list to find commonly searched locations, such as, Reykjavik. Figure on the right illustrates the use case of analytics server.

The team decided to skip the implementation of the Admin page (see figure 2) due to time constraints and lack of interest from Vegagerðin.

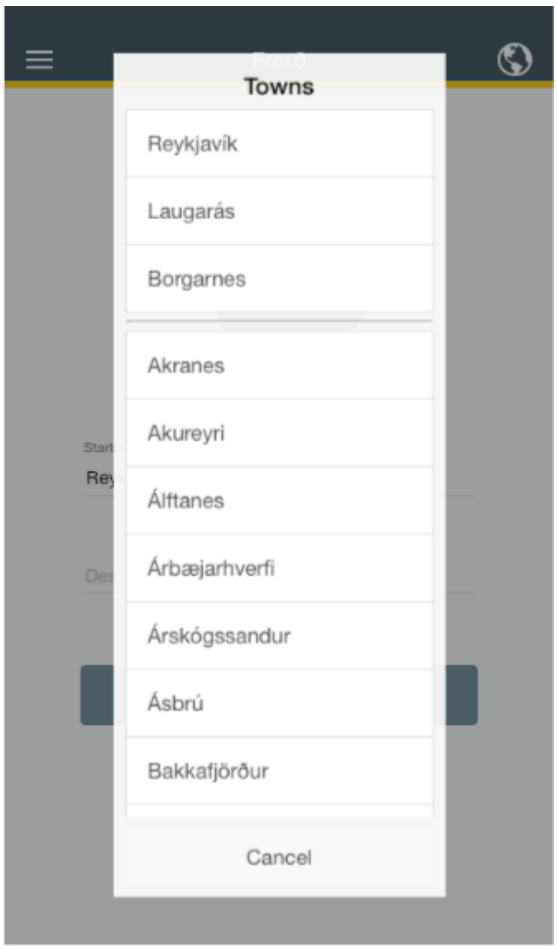


Figure 5: *Three most popular places*

5.3 User Interface Design

At the beginning of development, the team agreed on a user interface design that is roughly divided into three parts, a start screen, results which are displayed on a map and other options in tabs. The design of structure of the application has not changed over the course of development, but the team has made a continuous effort to make the user interface as clear and easy to use as possible. Special consideration has been given to utilizing the available screen space as well as possible since this is a mobile application.

The color scheme of the app was selected after discussions with the product owner to fit in with other published material from Vegagerðin where the brand colors are charcoal, blue-gray and golden yellow.

On opening the app brings up the home screen. Its main function is to allow users to either select two places in Iceland or bring up a map displaying all conditions. In the beginning it was assumed that the logo of Vegagerðin would be included in the home screen but after deciding to use it for a launcher icon it was replaced with a map-shaped button to see conditions for the entire country.

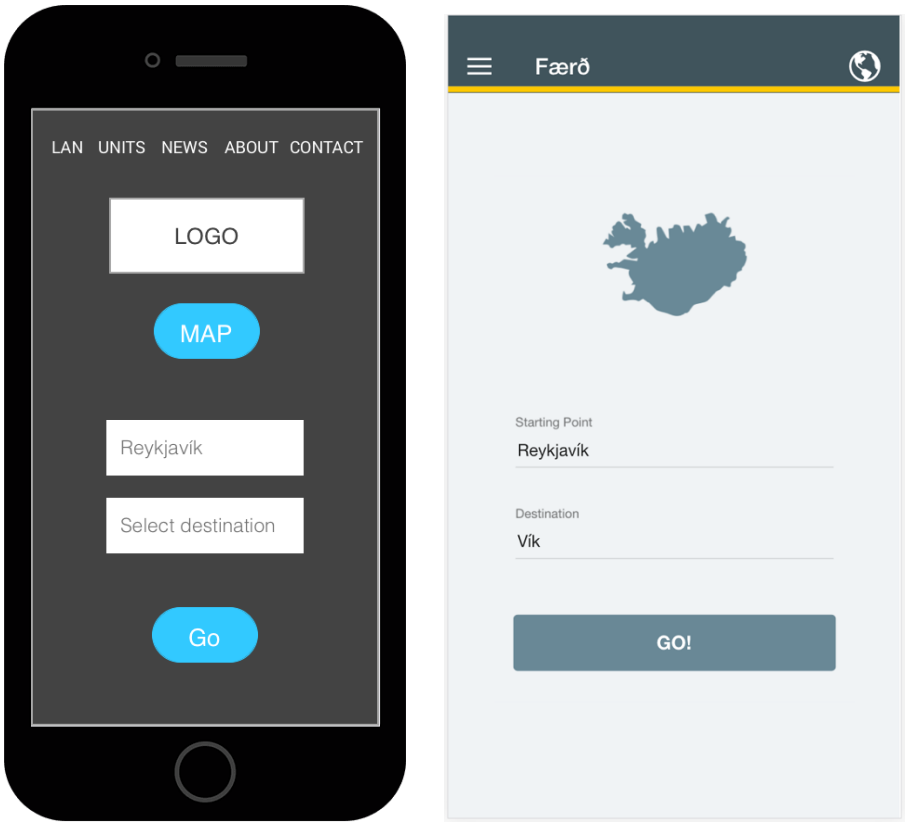


Figure 6: on the left: *Prototype of home screen*, to the right: *Home screen end result*

The language settings have been moved to a dropdown accessible by tapping the globe icon and other options to a side menu in order to keep the interface uncluttered. The News, About and Contact tabs have been moved to a side menu.

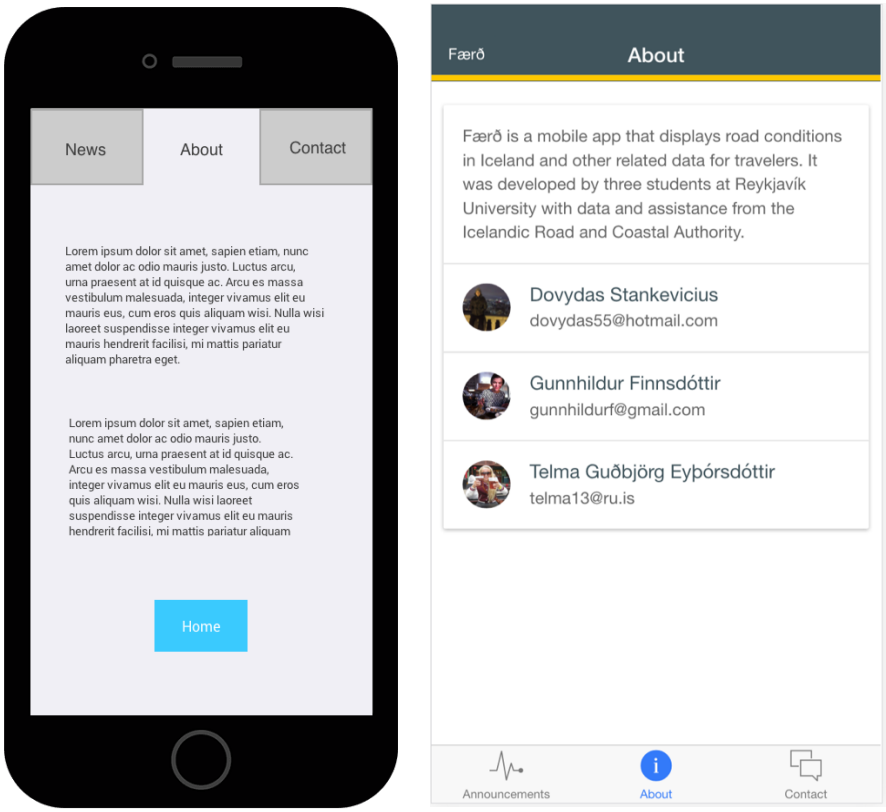


Figure 7: *Tabbed information screens*

After pressing Go on the start screen, the user is directed briefly to a splash screen that displays a photo appropriate for the season and when all data has been collected and parsed, the user will be directed to the results screen. This has remained unchanged throughout development.

The results screen consists of a map where the shortest routes between the two places that the user selected is drawn in color codes that map to different road conditions. The banner at the bottom is dynamic and changes according to what information is being displayed. The most important changes to the interface over the course of the project are that the options for what is displayed on the map have been moved to side menu and the banner at the bottom can be hidden to allow the map to take up the entire screen.

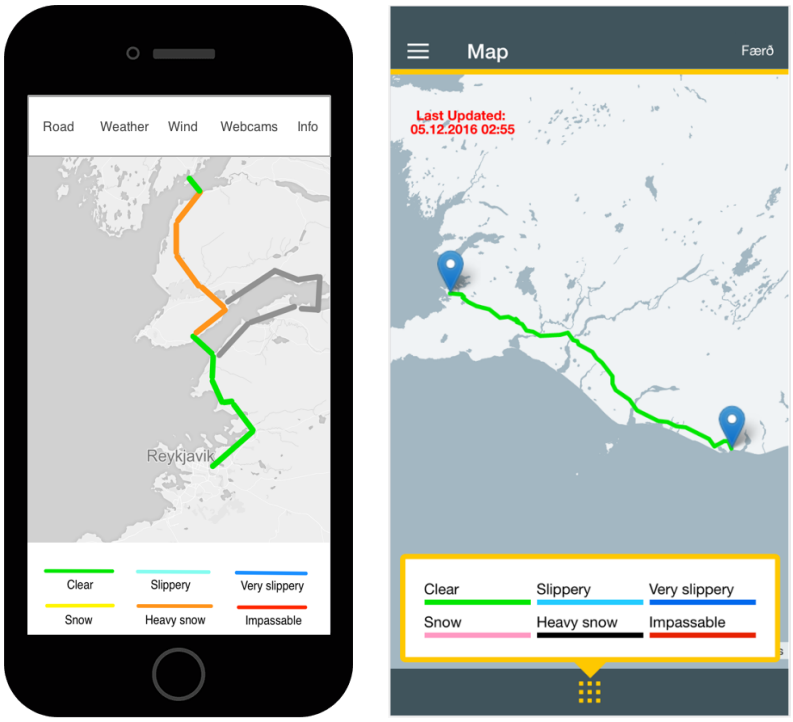


Figure 8: on the left: *Result screen prototype,* to the right*: result screen in the app*

Weather: The route stays on the map, but clickable weather icons appear in addition. The legend explains that they can be clicked to access information about the weather at that point in the route. When a user clicks an icon, the legend displays the information for that weather station.

Wind Gusts: Sections of roads can have wind gusts that are especially hazardous to truck and bus drivers and people travelling in recreational vehicles. By selecting this view, icons that display color coded icons for wind gusts appear on the map in addition to the route. The legend shows the user what these color codes mean.

Webcams: Vegagerðin maintains webcams along the entire network of roads in Iceland. When a user selects this option, icons for cameras along the route appear. The legend explains that the icons are clickable. When a user clicks one, thumbnails of available photos appear on the legend.

Information: The map shows any weight restrictions along the route and the legend displays the distance between the two places and an estimate of the time it will take to drive and how much it would cost on average.

These options have been developed according to original design and are displayed in the user manual in chapter 10.

On the whole, the original design has been implemented with some changes that enhance the appearance and usability of the app, in addition to making it fit into the design standards required by Vegagerðin.

6. Quality Assurance

6.1 Programming Disciplines

For keeping coding style consistent between the developers, the team used JShint to validate the code before it got pushed to production. Editorconfig file was used for synchronizing editor settings between the development machine (i.e. use LF line endings, UTF-8 encoding, etc.). Every API endpoint deployed to the production has at least one load and acceptance test.
In order to maximize productivity, the team decided to avoid branching as much as possible for several reasons. First, to not waste time during conflict solving. Second, to make continuous delivery (CD) easier as it only tracks the master branch. In order to avoid complex conflicts, the team decided to make it a rule of thumb to commit code as frequently as possible, at least once a day, with descriptive commit messages. Finally, another rule of thumb was that if developer was not able to integrate his/her code by the end of the day (failing tests), everything until the last commit was reset and developer would start implementing the same feature from scratch the next work day.

6.2 Documentation

For maximizing the future maintainability of the source code, JSDoc style comments were used on the client-side while apiDoc style comments were used on the backend. JSDoc is used on the client side because it is a standard tool for documenting the source code, while apiDoc is oriented at API endpoint documentation. apiDoc is used to generate documentation for our API (faerdapi) as a part of automated build process. Generated documentation is automatically pushed to production for updating the API home page. Since the client (faerd-client) is proprietary software, the source code and its documentation is not openly available online. Therefore, all of the documentation generated by JSDoc is only locally accessible on the development machines. JSDoc generates all documentation automatically as a part of continuous integration.

6.3 Deployment time

To evaluate the efficiency of our build pipelines and to determine if it needs any refactoring, we measured the average time it takes to push code through deployment pipeline for both the API and client.

6.3.1 Faerdapi

It takes on average three minutes for the Jenkins CD server to generate Docker image, perform quality check, execute acceptance, load tests and synchronize all of the production machines with latest Docker image. This shows that the build server is fairly well optimized and current build system does not require any major optimizations.

6.3.2 Faerd-client

It takes on average of two minutes for the Jenkins continuous integration (CI) server to build the app and execute all of the quality checks including the unit tests. This is satisfactory time so the build pipeline does not require any major optimization.

6.4 Continuous Deployment – server

We have implemented a continuous deployment build system for our backend. The CD is divided into four major groups: commit stage -> acceptance stage -> load stage -> production stage. Each stage automatically triggers another stage once it has successfully done executing. In commit stage, CD server pulls latest changes from the Version Control System (VCS) and runs code inspection (checks code style), unit tests and generates Docker image. Acceptance stage automatically executes acceptance tests against each API endpoint and makes sure that the latest changes have not broken anything and API still continues to function normally. The load stage makes sure that each endpoint passes the performance tests and is efficient enough for production. Finally, the production stage synchronizes all production machines with latest changes. The main advantage of this architecture is that it gives instant feedback about the health of the project and instantly helps to identify issues. For example, if the acceptance stage fails, this is an indication that a certain API endpoint is not returning expected data, or if contrastingly load stage fails, that is the indication that algorithms used are not fast enough and need revision.

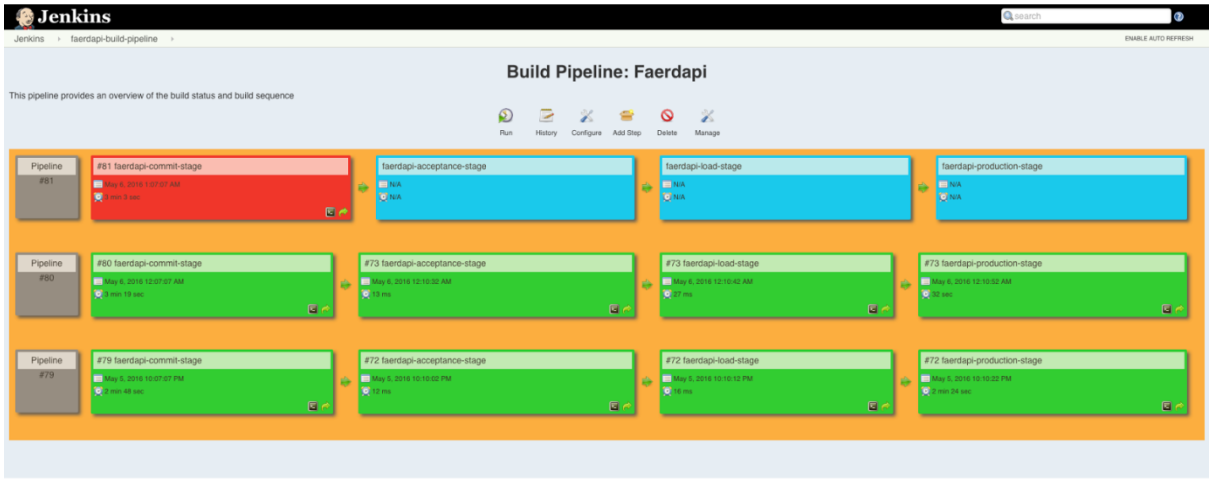


Figure 9: *Jenkins CD pipeline for faerdapi*

As it can be seen from in the Figure 1, build 81 failed at a commit stage. This is a quick visual feedback that the newly pushed code to the VCS did not pass quality check or failed some unit tests. Consequently, this commit has not been pushed to the production as indicated by blue boxes (blue boxes show stages where CD stopped and did not execute).

6.5 Continuous Integration – client

For the client we have implemented a continuous integration system. Since Vegagerðin does not want to deploy the app to the production yet, before making final revisions, therefore we

decided that it is sufficient to implement CI. Figure 2 illustrates build pipeline for faerd-client

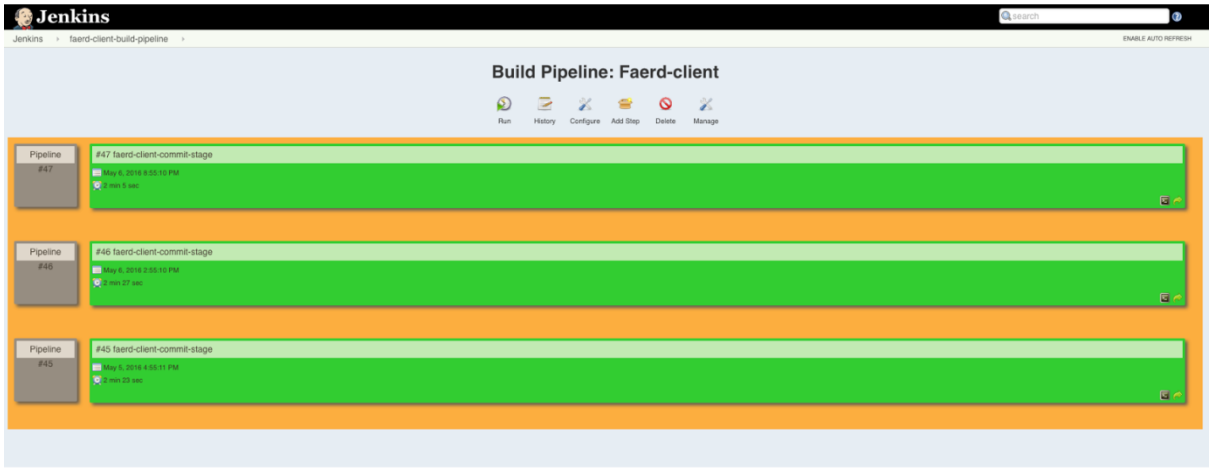


Figure 10: *Jenkins CI pipeline for faerd-client*

The continuous integration system has only commit stage. The purpose of the commit stage automatically pulls client code from VCS, downloads and installs required packages and plugins, runs code inspection and executes all of the unit tests.

For providing a visual feedback of the project, Jenkins generates graphs

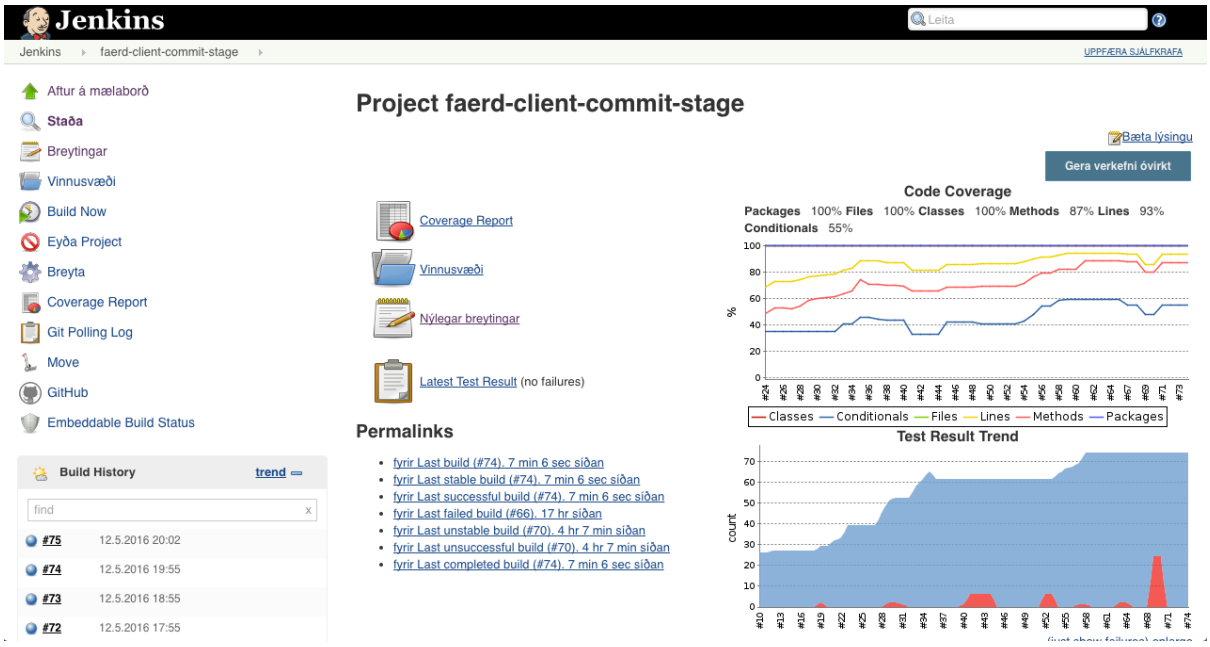


Figure 11: *Jenkins CI commit stage*

6.6 Security

Since publication of this software for general public use is planned, creating a well secured system was a major concern from the beginning of this project. Ionic engine is based on JavaScript and runs very similar to a web browser. Therefore, common attack strategies have to be kept in mind when developing the app. For example, the app UI is designed with security in mind. For general app use, user is allowed to interact with the program by

tapping on various icons or selecting data from a predefined list, thus minimizing the user input. However, where the user is allowed to type something into the app (contact form), official angular sanitize library is used for validating all of the user's input. This automatically removes all dangerous html strings from the input, thus minimizing the possibility of a cross-site-scripting (XSS) attack.

Since the backend heavily relies on caching for avoiding repetitive calculations, lots of attention has been dedicated for avoiding cache poisoning attacks. All the cache keys on the server are sanitized and validated by comparing them against a valid list of keys. Furthermore, all cache keys are hashed, thus making sure that all keys are standardized (same length) and not wasting memory. The caching on proxy servers and load balancing between production machines is very effective strategy for protecting against a small scale denial of service attacks. Also, every production machine runs a clustered node server, meaning that it runs a separate instance of a server on each CPU of the production machine. Every incoming connection is sent to the master node which internally load balances the requests between the child processes. Hence adding a second layer of load balancing and a second line of defense against a DdoS attack. Finally, this clustered node architecture makes sure that the server is always kept alive. Meaning that master node is always tracks the child processes and if suspicious/unpredictable behavior is detected, child process is killed immediately by the master node and new instance is forked, hence always keeping the server alive and healthy.

All of our production servers are protected by Vegagerðin firewall, meaning that it is impossible to access them via ssh outside the Vegagerðin private network. We have also added authentication to our Jenkins build server. The connection to the build server uses SSL encryption and uses official SSL certificates issued by Vegagerðin.

The future security optimizations include adding DNS load balancing between load balancers, which would add yet another layer of security against DdoS attacks. Additionally, we would install intruder protection software (IPS) on the production server to strictly allow the traffic to be forwarded exclusively by load balancers and reject all request that directly try to connect to production machines.

7. Testing

7.1 Unit Testing

One of the key features in our design is flexibility and to be able to make our system as flexible as possbile we need to have good unit testing for each functionality. By having good code coverage this enables us to reduce bugs in new features and the cost of change. Not only does it improve the code we have written, but also it forces us to write more testable code.

7.1.1 Client

For unit testing on the client we are using a test runner called Karma with the test framework Jasmine and to keep a track of our code coverage we are using a Jenkins plugin, Cobertura to generate an html for a visual progress. Our goal was to have, at all times, at least 70% code coverage and to force that we have generated in our continuous integration (CI) that no build passes without 70% code coverage.

Figure below shows code coverage for the client in our Jenkins CI.

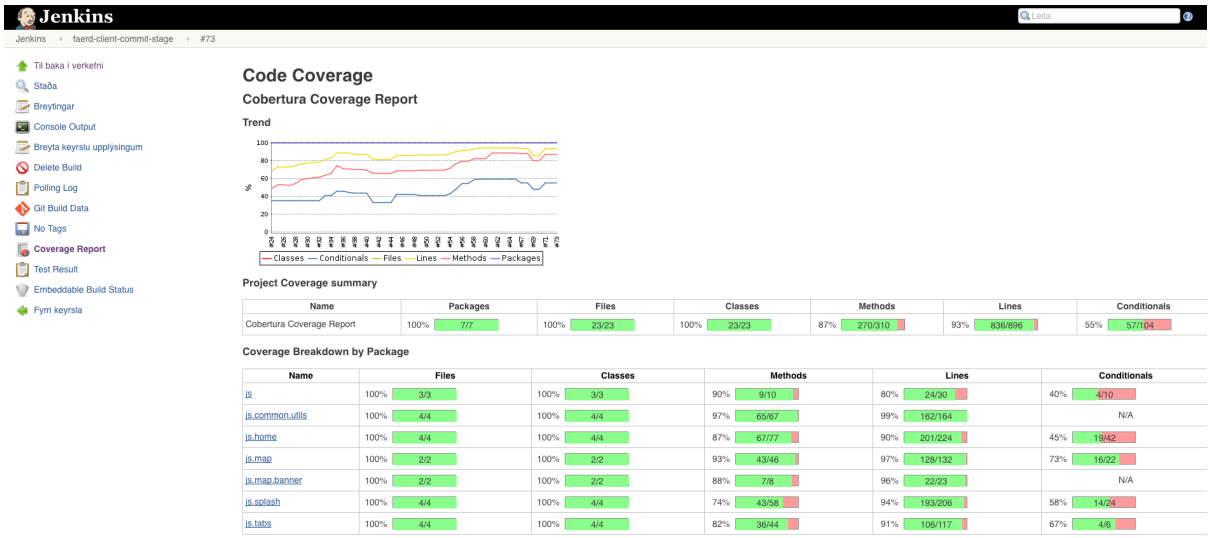


Figure 12: *Code coverage for client*

7.1.2 Server

To unit test the API we used Mocha test framework. For the API, only business logic has been tested. Other parts of the code, such as, server setup, connections to the database, etc. do not need to be tested using unit test strategy as they are tested via acceptance testing (end-to-end testing).

7.2 Acceptance Testing – Server

Acceptance tests, also known as end-to-end tests, test that the API is returning data and everything is functioning as expected. Every end point of the backend has a corresponding acceptance test. This testing strategy makes sure that data from the database is retrieved and that the other APIs, that faerdapi depends on, returns data. Finally, acceptance tests work similar to smoke tests - checks if every endpoint on the API is up and running. To set up the acceptance test on the backend, mocha test framework was used with combination of supertest testing module.

7.3 Load Testing – Server

Load testing test the efficiency of algorithms behind each API endpoint and help to determine the maximum load that each API endpoint is able to withstand within certain time frame. Furthermore, as load tests simulate a small scale denial of service attacks on the server, it helps to expose any obvious security vulnerabilities. To set up this testing strategy, mocha test framework was used with combination of supertest testing module.

7.4 Known Bugs

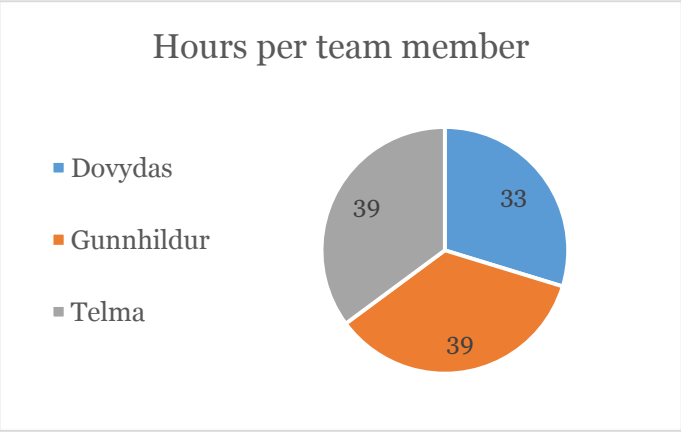
1. Single tap on a map cannot be detected on devices running iOS 9.3
   This issue could not be resolved as it is a result of a bug in Ionic framework. This bug will hopefully be fixed with upcoming updates in Ionic framework.

8. Progress

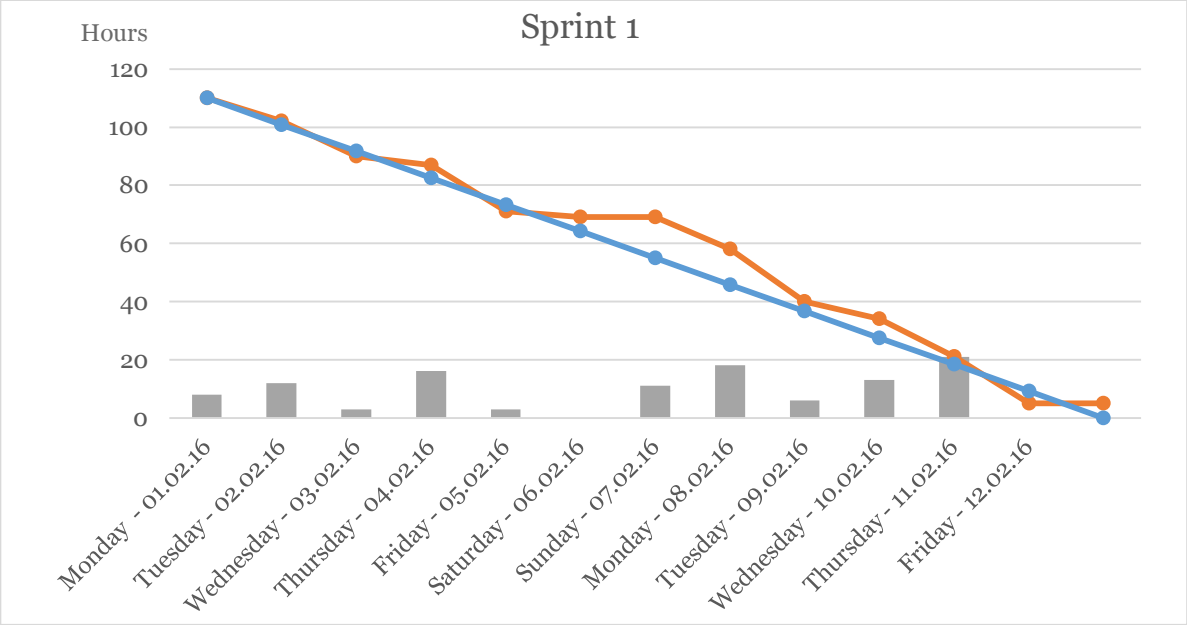
8.1 Sprint 1

During our first official sprint, we spent a combined 110 hours. During the sprint, we managed to complete all of the tasks that we committed to completing in our sprint backlog. As this sprint is at the very early stage of the product development, we spent most of our energy on report writing and planning. Additionally, we spent a great deal of time designing and discussing the UI of the app. In order to be aware of potential risks that might come up during the development of the product, we have discussed and assessed potential risks that might come up. As all of the team members are new to the React Native framework, we spent approximately 45 hours combined researching and learning the basics of the framework. Finally, towards the end of the sprint we managed to find time to create the server skeleton and set up the initial build components. Examples include writing the server build script (Grunt), creating the Docker container that is going to be used to wrap up the server and creating a Vagrant script for synchronizing the development environment among the developers.

At the end of the sprint we held a sprint retrospective and a sprint review. The sprint review was a quite informal meeting, where we presented to the product owner what we had achieved during the sprint. During our sprint retrospective we discussed the issues that came up during the sprint and reflected on how we could avoid them in the future. Examples of improvements that we decided to make is categorizing the hours spent during the sprint (development, research, etc.). Since we felt like we needed more organization and stricter policy regarding the work hour book keeping, our Scrum master took a responsibility to contact each group member daily, record the hours spent and categorizing them appropriately. Since our product documentation was lacking, we committed to revising it during the upcoming sprints before the second status meeting. Generally speaking, we are very happy with the sprint and we are quite happy with the current status of the project.



Graph 1: *Hours per team member during sprint 1*



Graph 2: *Progress during sprint 1*

8.2 Sprint 2

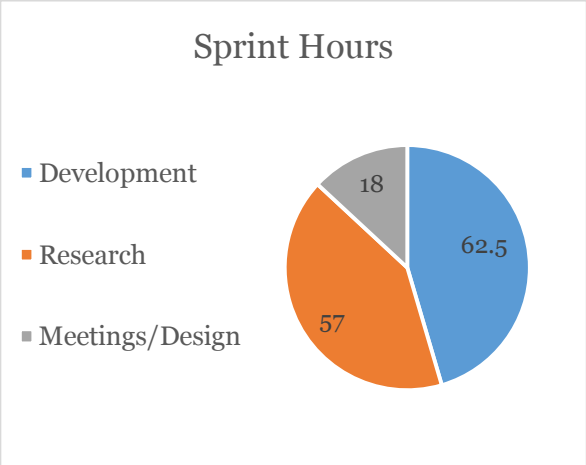
During this sprint we worked on both backend and on the client. On the backend, we started by creating MySQL database. The next step was setting up the Node cluster. Since we want our production servers to utilize the maximum potential of the host machines, we designed the server architecture to count the number of CPUs on the machine and fork itself accordingly. In order to improve server debugging, we created a custom logger which is controlled by setting an environment variable that controls different verbosity levels of the server. Additionally, we created a "/towns" endpoint where the client can fetch and modify all towns located in Iceland and get their coordinates. Finally, we set up the Grunt build to run all of the acceptance tests and load tests (using Mocha framework for testing) that we wrote for the "/towns" endpoint.

On the other hand, as we have less experience working with React Native we started small. We started by designing the project skeleton. Then we managed to run the base project on both Android and iOS devices. We integrated a simple HTTP client for fetching data from our server and saving it into a Realm database. Finally, it is worth mentioning that we did not manage to complete all of the tasks that we committed on completing on the sprint backlog. We started working on auto completion, however we did not finish it as we realized that we need to implement a custom component (we could not get to work any library). Therefore, we decided to continue working on this task during our next sprint. Finally, we did not even start implementing focus shifting and error handling. The primary reason why we did not manage to complete all of the tasks is due to our lack of experience with the React Native framework. Nonetheless, we are hoping to become much more efficient as we will gain more experience.

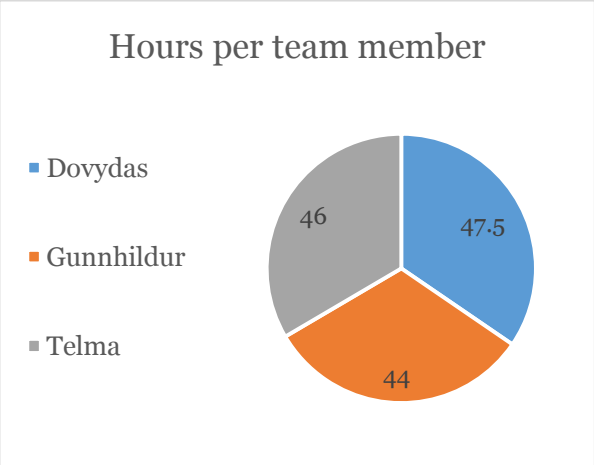
By the end of the sprint, we held a sprint review and sprint retrospective. During the sprint review, once again we showed everything we had accomplished to the product owner. During the sprint retrospective we discussed on possible testing strategies for making sure that everything is working on both Android and iOS as we experienced some inconsistency

between the platforms. One of our solutions was creating an continuous integration server for executing Android and iOS builds. Furthermore, during our app development, we discovered that certain React Native packages only work with specific versions of Node (some packages require latest version of Node, which we are avoiding as it is unstable version). During the retrospective meeting we assessed this risk and discussed its impact on the project.

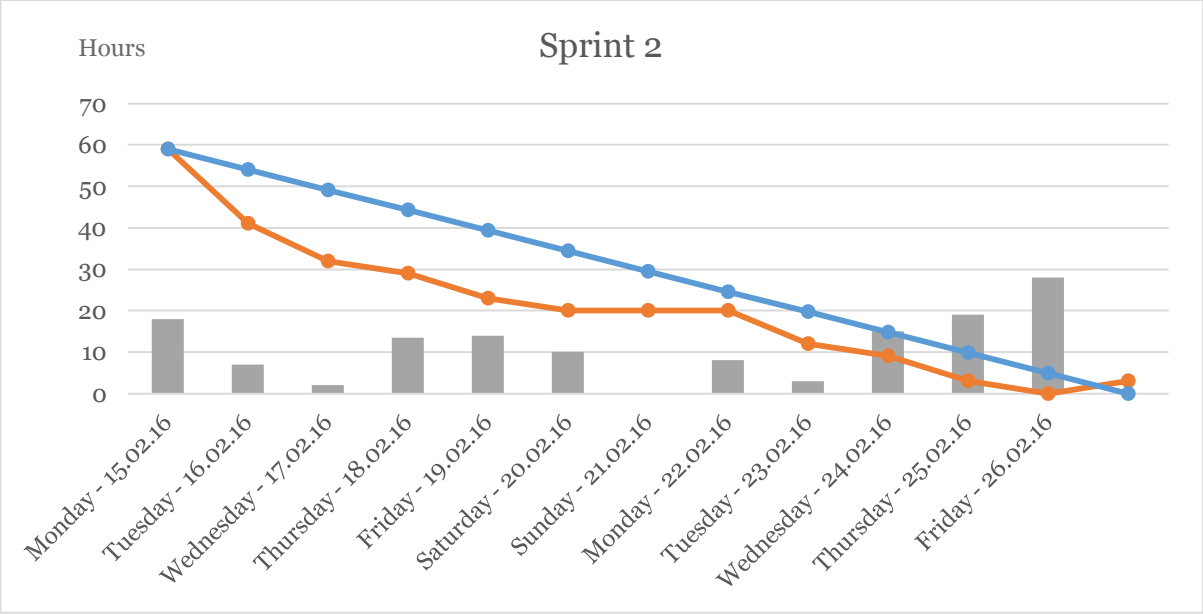
Finally, the major problem we experienced during this sprint was that our continuous delivery server got hacked. We made our build server accessible to the public (did not require any authentication) and consequently after couple of days the server got hacked and all of our production machines including build and test machines got destroyed. Consequently all of the work that we spent setting up the continuous delivery have to be redone in the next sprint. During the retrospective meeting we assessed the damage done and discussed what security measures should we follow in the future to avoid these accidents. To conclude, we believe that we are on track with the project even though we experienced couple of hiccups during this sprint.



Graph 3: *Division of hours in Sprint 2*



Graph 4: *Hours per team member in sprint 2*



Graph 5: *Progress during sprint 2*

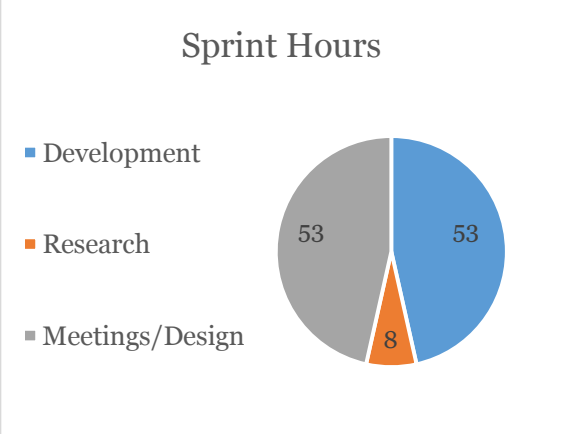
8.3 Sprint 3

During this sprint we spent a combined 114 hours. We were under the estimated sprint capacity of 120 hours, because some of the team members had very high workload in other courses or had personal distractions. Nonetheless, we managed to add a lot of new functionality to the backend. Examples include returning all webcams and weather stations on a custom route. We have also written acceptance and load tests for both of the endpoints. We have added a tool for automatically generating API documentation as a part of the build process. All API documentation is generated in an HTML format and the API manual page gets updated automatically when new changes get pushed to the VC. Probably one of the most important tasks that we completed throughout this sprint is recovering the build pipeline after the hack. As mentioned earlier, we had to redo the build server from scratch. In order to reduce the risk of future security breaches, we have added a basic authentication and encrypted the connection (SSL) to the Jenkins control panel. The current status of our continuous deployment pipeline is that every code that gets pushed on the master branch on VC gets pulled by the build server, the unit tests/acceptance tests/load tests get executed automatically and upon successful test passing, new code gets automatically deployed to the production machines.

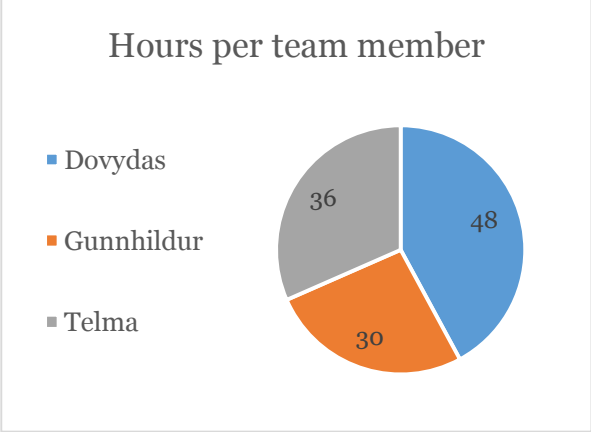
On the client we finally finished autocomplete by implementing a custom component. We have also implemented a complete routing stack. Furthermore, we have revised significantly the code, making it more modular. We have implemented a mechanism for loading essential data from the database on the app startup. Finally, there are features that we did not managed to complete from our sprint backlog. Nonetheless we will try to implement them during the upcoming sprint.

As always, by the end of the sprint, we held both sprint retrospective and sprint review. During the sprint review we assessed and investigated in detail the causes for the security breach and discussed security measured that would help us to avoid it in the future. During the sprint retrospective the issues brought up were better sprint planning and the importance of all team members being present during the scrum ceremonies. This had negative effect on our sprint organization as one of the team members were not present in

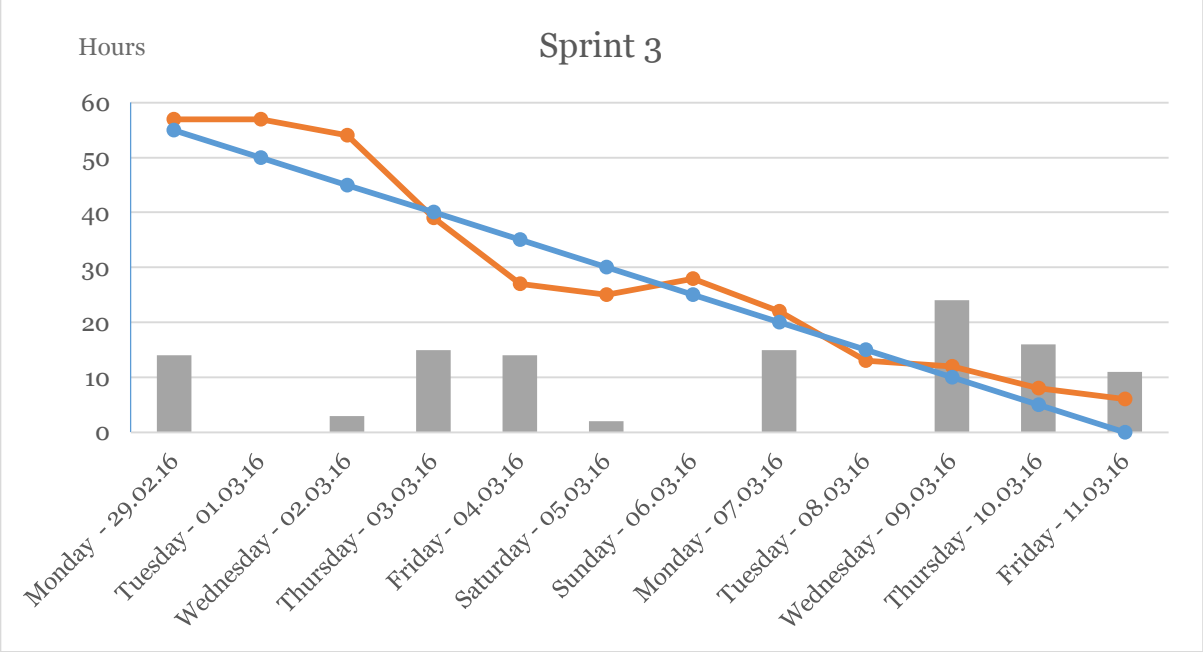
the sprint planning due to an exam. We agreed upon the fact that in the case of a team member not being present in an essential ceremony like sprint planning, we should reschedule the meeting so everyone could show up. Another issue brought up was the need for better reporting to the scrum master the time spent, specifically not only number of hours, but also tasks worked on and their type.



Graph 6: *Division of hours in Sprint 3*



Graph 7: *Hours per team member in sprint 3*



Graph 8: *Progress during sprint 3*

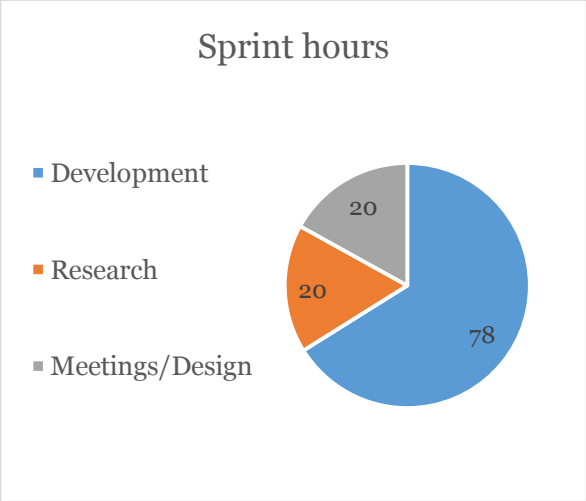
8.4 Sprint 4

In our fourth sprint we started to focus more on the client since the server was coming together. Our biggest feature this sprint was setting up the language support for the app. We also implemented an auto detect for current location. Although this sprint was mainly focused on the client we also made some security updates on our server. Since our security breach in sprint three we started to think more about protecting the server from attackers. In our Redis database we started hashing and sanitizing the keys to prevent hackers to input malicious code, hence preventing cache poisoning attacks.

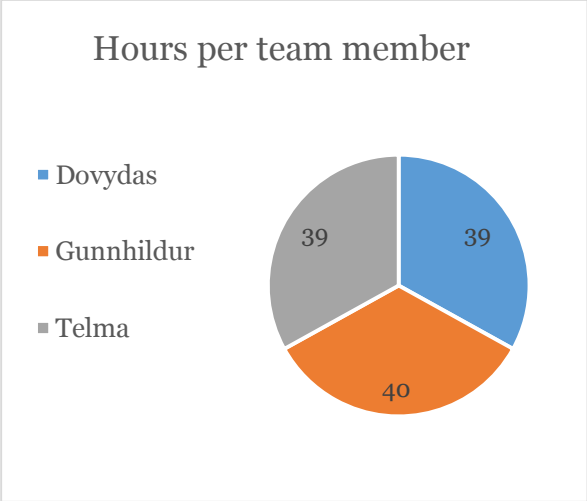
One of the biggest improvement in this sprint is that we started to use a scrum board for all upcoming tasks relevant to that sprint. It helped us make our improvements on the project more visible and to keep a track of what tasks were already in progress. The team also improved a lot in reporting hours they spent each day and on what task.

In our retrospective meeting we still thought that we needed to keep on improving in our planning, although the scrum board made us a lot more organized we still felt little confused about how the tasks were divided between us so we wanted in future sprints to be more clear on what task each member is working on.

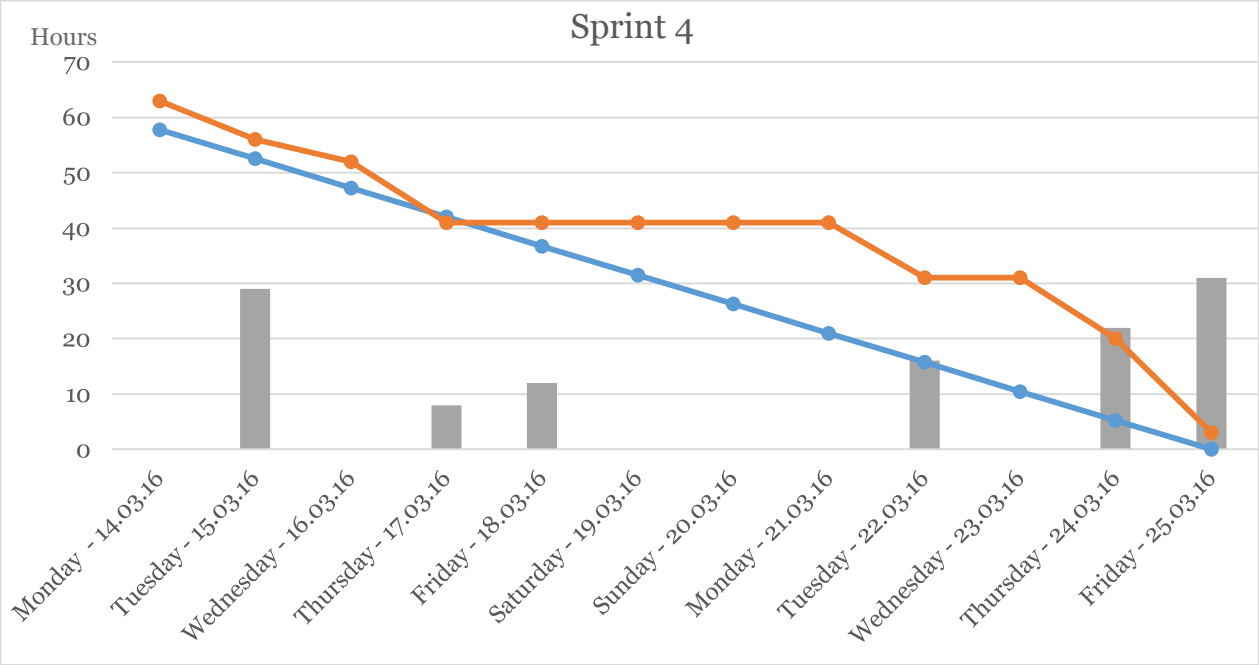
We spent a total of 118 hours which was again a little under the estimated time which was due to the fact that one of the members had a medical emergency.



Graph 9: *Division of hours in Sprint 4*



Graph 10: *Hours per team member in sprint 4*

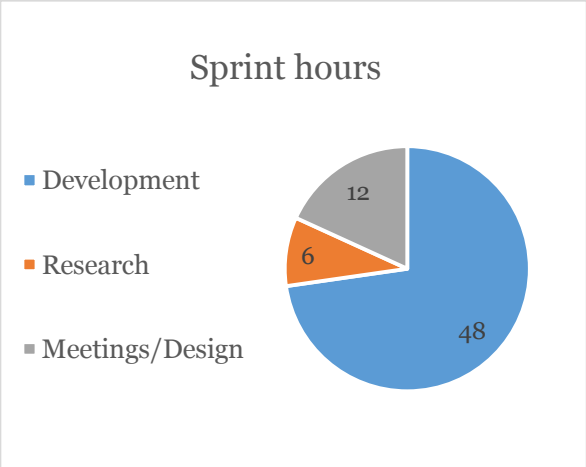


Graph 11: *Progress during sprint 4*

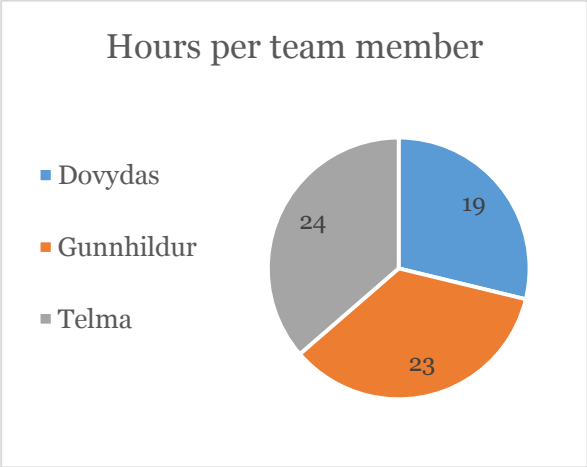
8.5 Sprint 5

During sprint five we all had finals in our other subjects so we decided to lower the capacity from 120 hours to 60 hours, though we ended up spending 66 hours. Although we had exams we managed to setup the structure for the map as well as routing from home screen to map through splash screen.

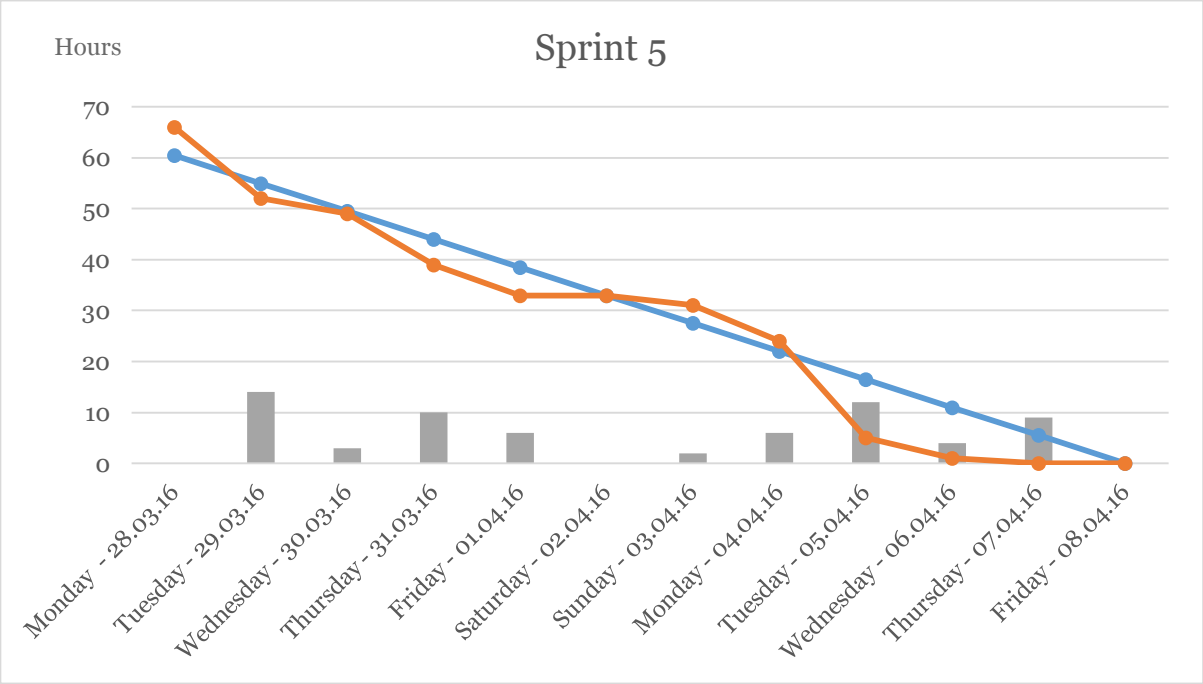
In our retrospective meeting we addressed that we needed to keep a live document for our risks throughout the rest of the project due to the fact that we encountered risks in the previous sprints that we had not predicted in the start of the project.



Graph 12: *Division of hours in Sprint 5*



Graph 13: *Hours per team member in sprint 5*



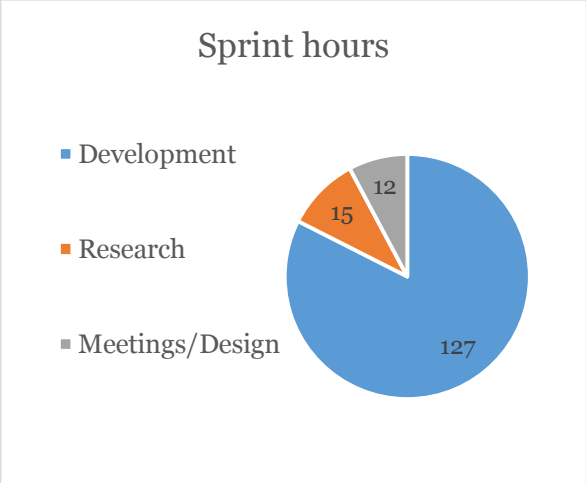
Graph 14: *Progress during sprint 5*

8.6 Sprint 6

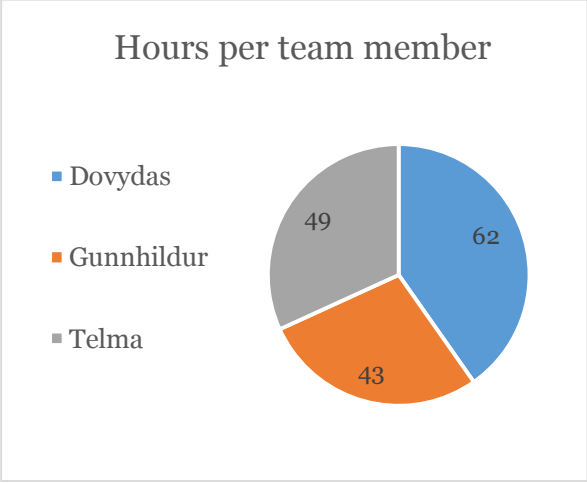
From the start of the project we knew that there were risks in making the client purely in React Native, since it is such a young framework, so we always kept in mind that if we had too many obstacles we would switch over to Ionic, which would be no problem since all members in the group have really good experience with AngularJS. In sprint 6 we decided to have a meeting about our progress with the client so far in React Native and after weighing the pros and cons we decided to switch over to Ionic. This did not mean that our previous work had gone to waste because a lot of it was reusable and most of our time in the previous sprints had been focused on the backend.

We spent sprint 6 mainly on moving over to Ionic and since all of the members had finished their exams we could spend a lot more hours on this project. Among with moving what we already had in our previous project we also managed to start returning conditions for roads along with relevant weather data and webcams for all Iceland. We also implemented a splash screen that changed according to relevant season. We spent a total of 154 hours and we managed to complete all of the tasks that we committed to in the beginning of this sprint

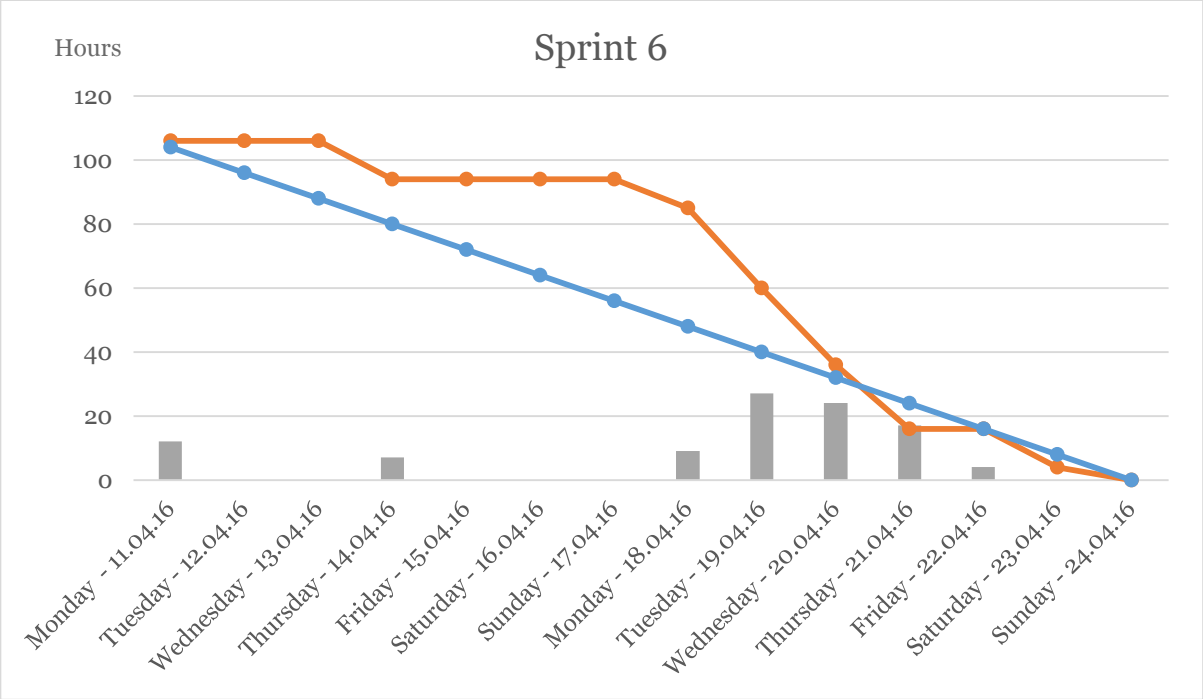
In our retrospective meetings we decided that we needed to spend more time in each sprint to reflect on previous sprints and updating the reports each week. We also wanted to have more scrum meetings with the team at Vegagerðin to get constant input on our progress throughout the project.



Graph 15: *Division of hours in Sprint 6*



Graph 16: *Hours per team member in sprint 6*



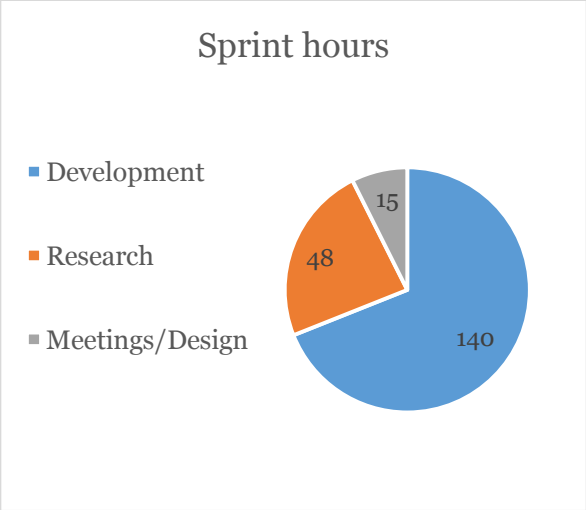
Graph 17: *Progress during sprint 6*

8.7 Sprint 7

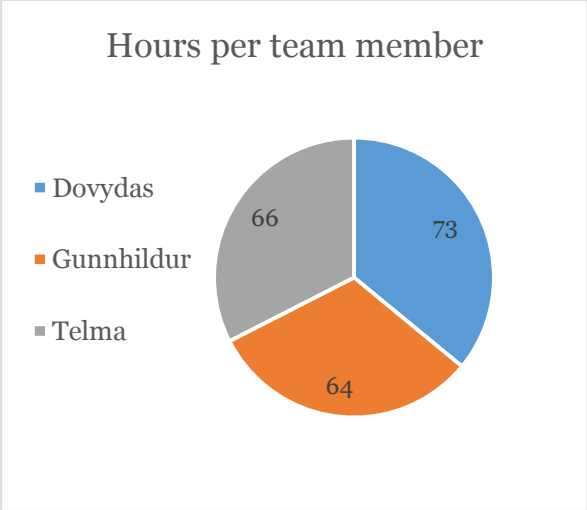
In sprint 7 we could see right away that we made a right choice by switching over to Ionic. We made way faster progress and each task took less time since all of us have a really good knowledge in Angular. We implemented a dynamic banner on the map that displayed information according to the icon that was clicked. For example, when you click a camera icon you get thumbnails for that specific camera in the banner. On the server side we made an API for a custom route which the whole team did together.

In each sprint we had become more and more organized as a group and one of the things we started doing in sprint 7 was to register issues on GitHub. This helped us keep an even closer track on the tasks we had left and the importance of each of them.

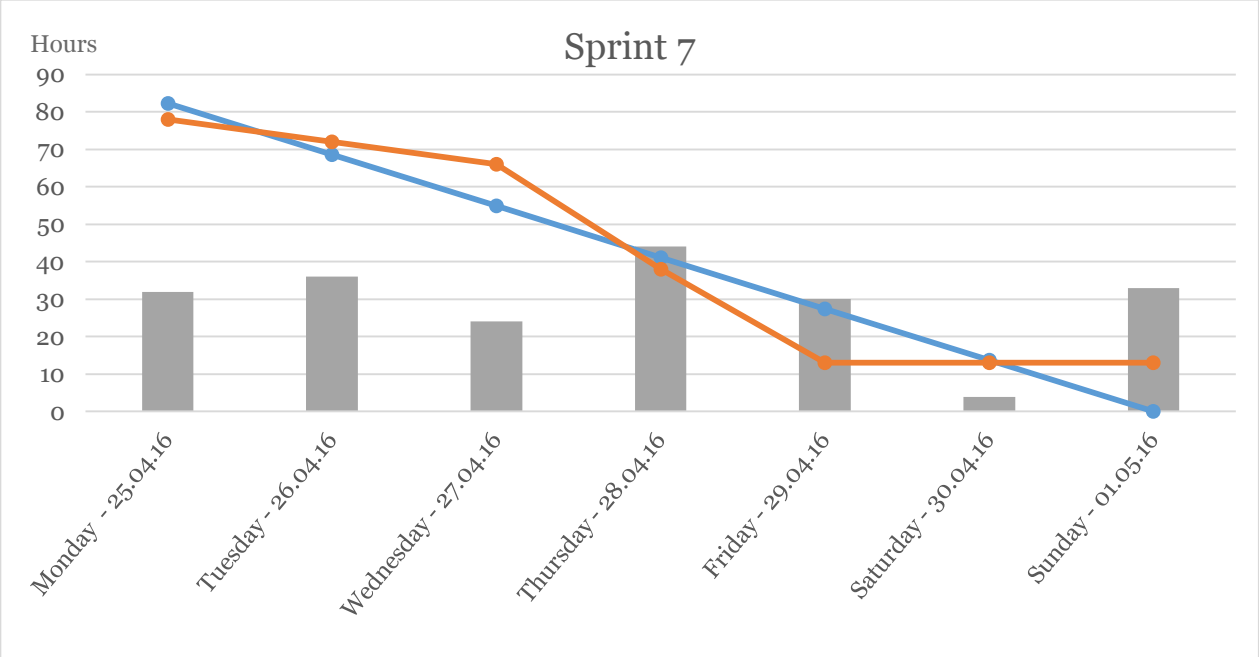
In our retrospectives we discussed that we should continue to meet every day and work in the same environment since it produced the most productivity we also decided to dedicate next sprint to unit testing and report writing.



Graph 18: *Division of hours in Sprint 7*



Graph 19: *Hours per team member in Sprint 7*



Graph 20: *Progress during sprint 7*

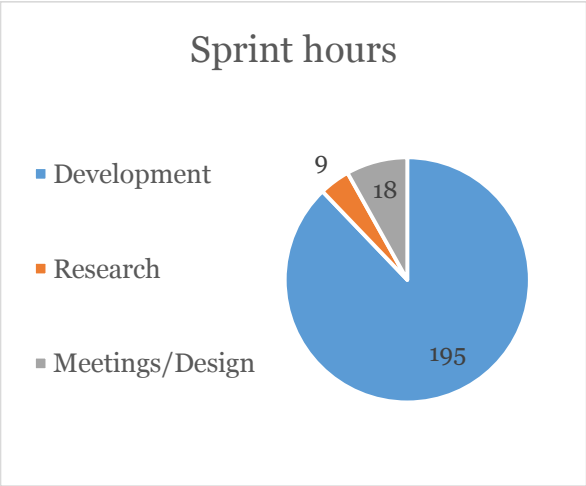
8.8 Sprint 8

Since the deadline was approaching, and the progress on the client side was going really well, we spent most of our time writing the reports and doing unit testing in sprint 8. We started the sprint by setting up libraries for the unit testing on the client.

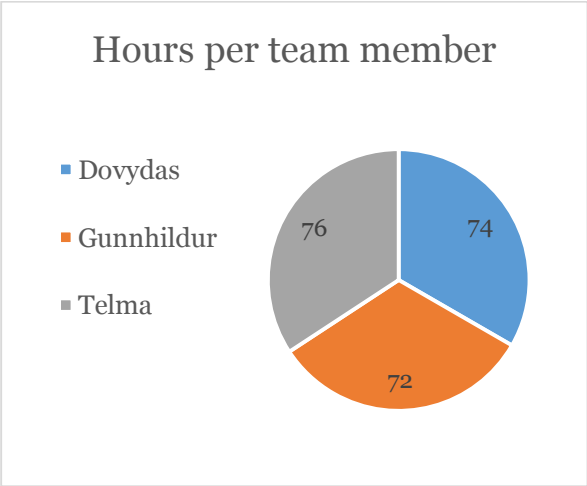
Along with the report writing and unit testing we also managed to do a lot on the client side. We addressed the issue that iPhone's do not register a tap on the screen in Ionic, which is a known bug, so instead a user has to either double click or hold down to be able to click an icon. We also did a lot of styling for example updating the tiles on the map, make our own icons and error handling for the list input.

On the server we made an API for the daily announcements from Vegagerðin.

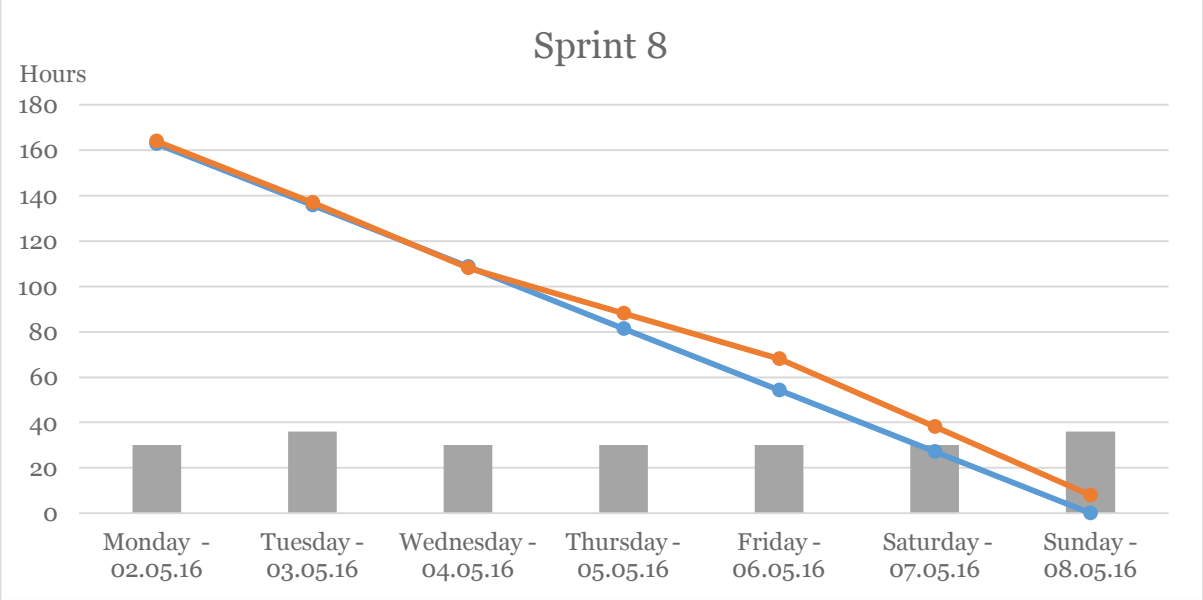
In our retrospective we discussed some of the points that came up during a meeting with Vegagerðin, for example we decided to incorporate some changes to the results screen that were suggested, for example adding a popup with a town name to the map and revising the color code for conditions. On the whole we are satisfied with the status of the project going in to the last sprint. Once again, working together as a team in the same place has proven to be effective and we will continue to do so in the last sprint.



Graph 21: *Division of hours in Sprint 8*



Graph 22: *Hours per team member in Sprint 8*



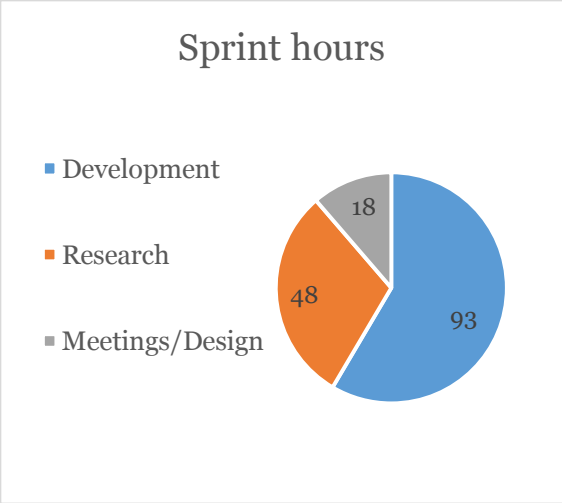
Graph 23: *Progress during sprint 8*

8.9 Sprint 9

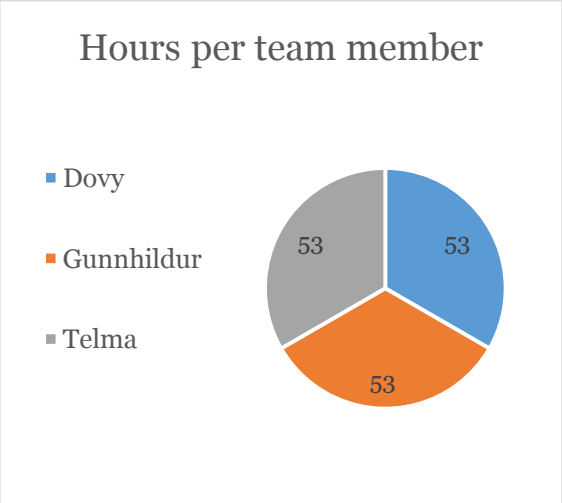
During the last sprint, we barely did any development. Instead, most of the time was spend on doing the final style changes, improving the localization by adding more languages and increasing the code coverage. As implementing the analytic server was a very complex procedure, lots of attention was dedicated towards its completion. We had major difficulties setting up the Zookeeper and Kafka broker and inner linking them. This issue was caused by

Vegagerðin firewall. Nonetheless, with a help of Docker technology, we managed to wrap both zookeeper and Kafka into a single Docker container and successfully deploy it.

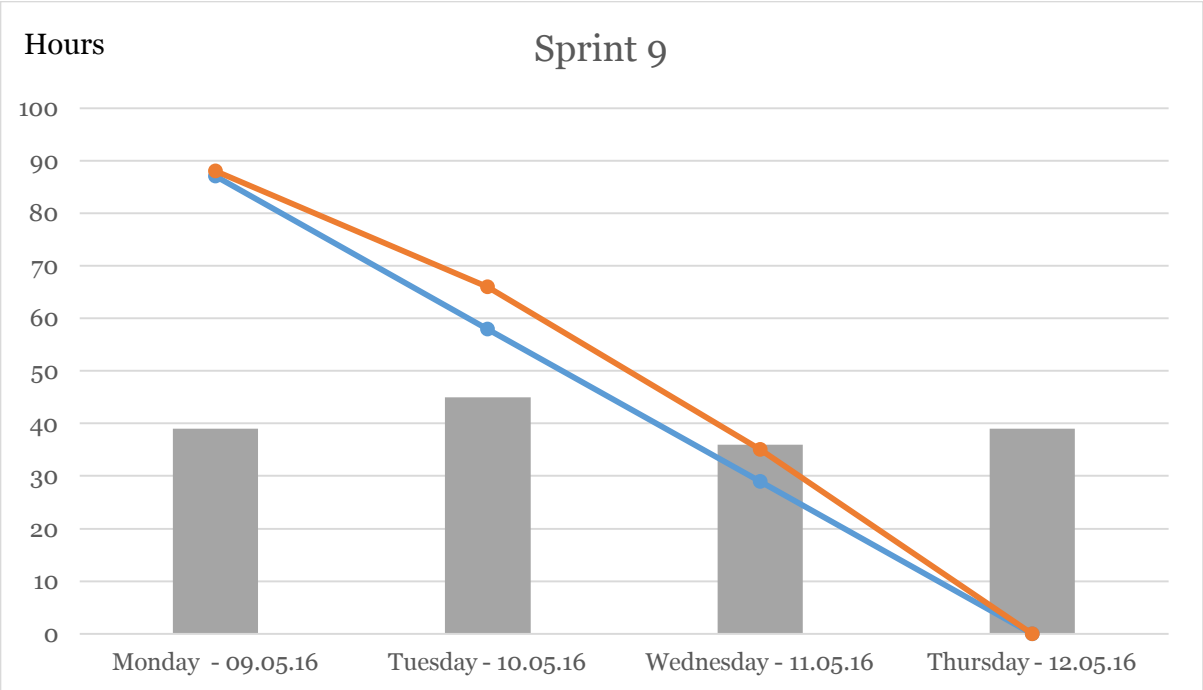
For the retrospective, status meeting 3 provided us with an excellent insight on how we could improve the final presentation. Also we got some useful pointers on the report and we spend the end of this sprint preparing for the final presentation and making final touches to the documentation.



Graph 24: *Division of hours in Sprint 9*



Graph 25: *Hours per team member in Sprint 9*

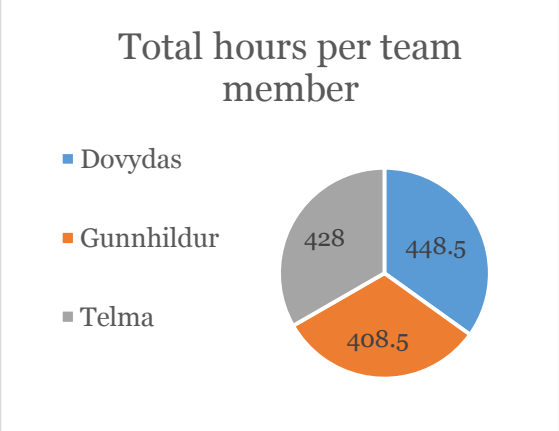


Graph 26: *Progress during sprint 9*

8.10 Sprint summary

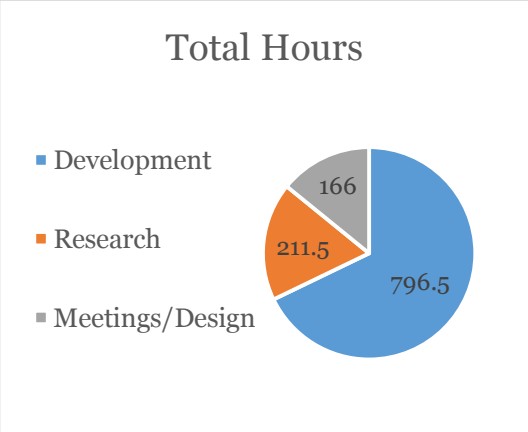
8.10.1 Time spent

At the end of sprint 9 a total of 1282 hours have been spent on the development of the app. Initially we estimated that the project would take 1148 hours so we have a 11,7% increase from the initial estimate. Considering that we switched development frameworks for the client and suffered some security breaches that crashed our server, we consider the additional time acceptable.



Graph 27: *Total hours per team member*

There is not a great difference in the time spent by individual team members over the course of the project and all members have exceeded the time requirements laid out by Reykjavík University for a final project. As we have spent the majority of development time working together in the office space provided by Vegagerðin there is less variation in the time spent by individual members at the later stages of the project.

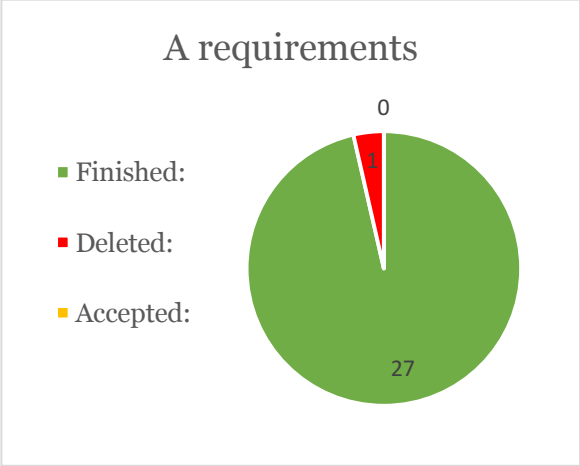


Graph 28: Total hours

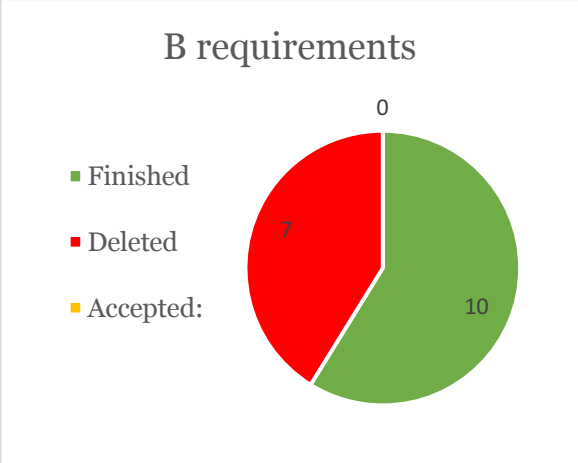
Around two thirds of the aforementioned 1282 hours were spent on development, which also includes documentation. It is often difficult to clearly differentiate between development and research during programming since these two aspects often go hand in hand. Much of the needed research took place before development started, both while building the prototype and during sprint zero so we have only logged 163 hours for research during this development cycle. The remaining time was spent on meetings among team members as well as meetings with instructor, status meetings for the project and meetings to discuss the project with Vegagerðin staff, including the product owner.

8.10.2 Product Backlog

According to our product backlog we have finished all of the A requirements except the ability to see alternative routes. We removed this requirement from the product backlog as we were unable to find a routing service that gave us a selection of routing options to prioritize according to condition. The complexity of implementing such a service exceeds both the time and scope of this project.



Graph 29: *A requirements*

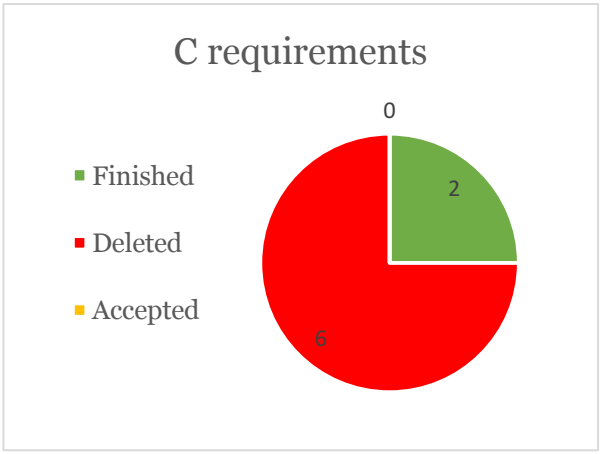


Graph 30*: B requirements*

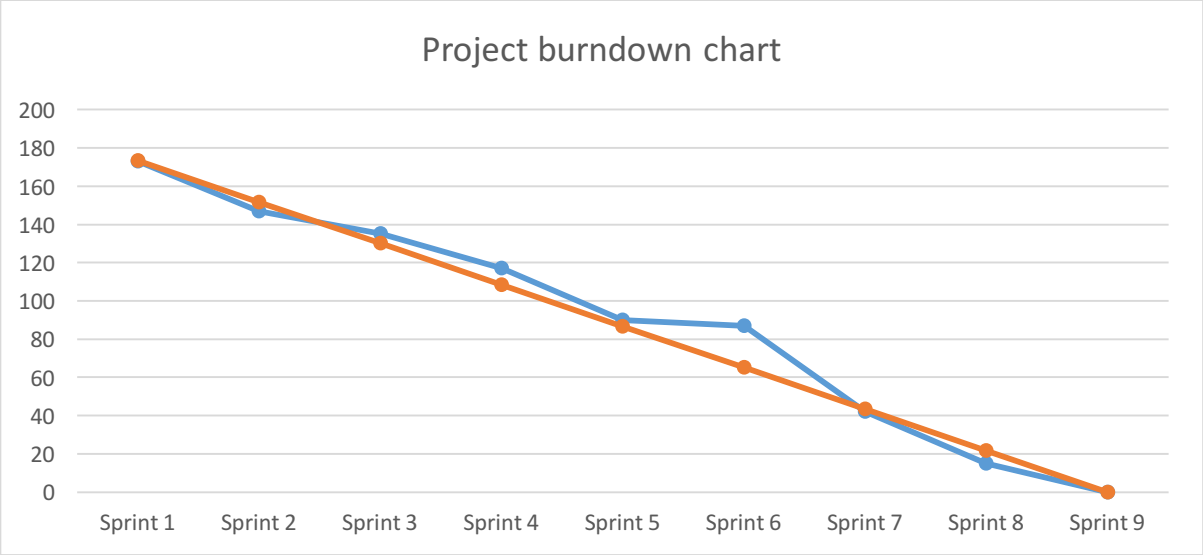
We managed to complete most of the B requirements except the ability to select two locations by tapping the map. Instead we replaced this feature by allowing the app users to select locations from a predefined list of towns. After closely examining the primary user groups, the ability to see road temperature was removed as well, because the team decided that such an information is irrelevant to the app as it would crowd up the UI. Due to the fact that the team decided to cancel the implementation of the admin page (after discussing this with Vegagerðin), consequently several other relating tasks had to be deleted from the product backlog. Hence tasks such as the ability to see comments submitted by the app users and seeing server traffic in real time were deleted.

As only Icelandic towns are stored in the database and the fact that they are also backed up in a local json file, tasks such as migrating database and backing up database were deleted as they were unnecessary.

We managed to finish some of the C requirements, however due to time constraints we had to delete several C tasks from the product backlog. Example of such a tasks are the ability to see traffic jams on a custom route, support for color blind users, specify plate number for getting more accurate gas costs and getting push notifications if approaching extreme conditions.



Graph 31: *C requirements*



Graph 32: *Project burndown chart*

9. Operations Manual

9.1 Client

The source code for the client can be found in zipped folder that was handed in, under a folder called faerd-client. The project is also hosted on GitHub [https://github.com/Faerd/faerd-client], however since this is a private repository, an access to repository would have to be requested.
Pre-requisites:

1) NodeJS - https://nodejs.org/en/
2) Git - https://git-scm.com/downloads

To prepare the development machine for development execute the following command script

[~ faerd-client]# ./scripts/init/client-init.sh

This script will install all of the necessary development packages, such as Ionic, Gulp, Karma, Cordova and Bower that are necessary for running and building the app. Do note that this script would work on any operating system. Once desired changes have been made to the source code, the app should be built using the following script

[~ faerd-client]~ ./scripts/build/build.sh

This script will run all of the unit tests, download and install all of the necessary ionic plugins and prepare the project for platform specific compilation.

Android build
Pre-requisites:

1) Android SDK - http://developer.android.com/sdk/index.html

Android can be compiled on any operating system that has Android SDK tools preinstalled (specifically tools for API 19). To build android run the following command

[~ faerd-client]~ ionic build android

This will compile the Android source code and generate *.apk file which can be used to deploy the app to any Android phone. To deploy the app to any Android phone running API 19 and above, connect the phone to a development machine using a USB cable and execute the following command:

[~ faerd-client]~ ionic run android

iOS build
Pre-requisites:

2) Xcode - https://developer.apple.com/xcode/download/

To be able to build an iOS version of the app, a computer running iOS with Xcode is required. To build the iOS version of the app, execute the following command

[~ faerd-client]~ ionic build ios

To deploy the app to any iPhone or iPad, connect the device using the USB cable, open the generated Xcode project, which can be found under *platforms/ios/Faerd*.xcodeproj and click play button which is located at the upper left corner of the project window.

9.2 Server

The source code for the client can be found in zipped folder that was handed in, under a folder called faerdapi. The project is also hosted on GitHub [https://github.com/Faerd/faerdapi], however since this is a private repository, an access to repository would have to be requested.
The API can be divided into three sections:

1) Development machines
2) Build servers
3) Production servers

Each section has to be set up and configured automatically by executing certain scripts as system administrator (root). Do note that all initialization scripts provided in this manual are guaranteed to work exclusively on machines running Ubuntu-14.04 LTS (this applies for build and production servers). If desired to set up this project on different Linux flavor, all installation has to be done by hand using provided scripts as guidance for required packages. It is important to point out that build servers and production servers must be deployed on machines running Linux, simply because Windows and OS X cannot natively run Docker containers unless third party virtualization software is used. All necessary scripts for setting up the environments can be found under scripts folder which is located at the root of "faerdapi" project.

9.2.1 Development machines

Our product heavily relies on virtualization technology; therefore, it is trivial to get new developers into development independent of the operating system of their preference (literally any operating system can be used for development). Before running the server locally on the development machine, the machine has to be pre-installed with following software:

1) Vagrant https://www.vagrantup.com/downloads.html
2) Virtual box https://www.virtualbox.org/wiki/Downloads
3) Git https://git-scm.com/downloads

Once vagrant has been downloaded and installed successfully, the programmer has to clone the Git repository and run the following command inside the root of the project

[~ faerdapi]~ vagrant up

This will create a virtual machine (Ubuntu 14-04 LTS) based on the image specified inside Vagrant file (which can be found at the root of the project) and automatically install all of the necessary packages that are needed to run the project. If developer has installed new package as a dependency and wants to sync all developer machines, the developer has to edit ***scripts/vagrant/provisioner.sh*** script and push it to the Git repository. This way all

developers are kept in sync with required packages and “works on my machine” syndrome is completely avoided. Hence, in simple terms, all developers share same virtual machine. Furthermore, vagrant script creates virtual file pointers as well as enables port forwarding between development machine and generated virtual machine. Meaning that all development can be done on the developer’s computer using their favorite development tools (for example their favorite text editors) and all of the new changes get compiled on the virtual machine instantly.
To run the API, first the developer has to ssh into the VM generated by vagrant. This can be done with the following command

[~ faerdapi]~ vagrant ssh

Then navigate to the project folder and run the node server

[~]~ cd src/faerdapi && node src/boot.js

This command will run the node server on port 8000 and API will be accessible at

http://localhost:8000

Finally to push the code into the production, the developer simply has to push the code to the git and the Jenkins build server will automatically detect new commits and roll out new changes into production as a part of the automated process.

9.2.2 Build server

To install and configure brand new production server simply execute the following script which will install all of the necessary packages and dependencies, such as, Git, NodeJs, Grunt, Jenkins and Docker that are necessary to run the project

[~ faerdapi]# ./scripts/init/ubuntu-14.04-build-server-init.sh

Among other necessary packages, this script will install and launch Jenkins build server which is used to control the entire build pipeline. Once the script has finished executing, Jenkins can be access on http://serverIP:8080 using any browser. Important note: this script will not add authentication to the Jenkins control panel. It is the responsibility of the system administrator to add their company SSL certificates, as well as, adding the usernames and passwords of the staff that has access to the build panel. Make sure to add access control immediately after the installation as this opens a severe security vulnerability as anyone will be able to execute malicious code on the build server via Jenkins control panel.
After adding the authentication to the build panel, the system administrator is required to create a DockerHub account. This technology is used for distributing the Docker image to remote production machines across the network. DockerHub account can be created at hub.docker.com. Once account has been created, the next step is to cache up Docker credentials inside the build server by running the following command

[~ faerdapi]~ docker login

Once Docker hub credentials are successfully cached up locally, it is time to configure Jenkins server to perform continuous deployment.
Inside Jenkins home (http://serverIP:8080) click on “new item”. Enter an item name “faerdapi-commit-stage” and select it to be a freestyle project. After Jenkins has created an empty project, click on “Configure” to configure the project. Under “Source Code Management” select Git and specify the URL where the project is hosted (in our case github). Additionally add credentials to the Git repository so Jenkins could pull latest changes from the version control system.
Second step is specifying build triggers. Enter the following cron expression under “Poll SCM Schedule”

H/59 * * * *

This will make Jenkins poll the version control system every hour and if any changes (new commits) have been detected it will automatically build the project and push latest changes to the production. Finally add a build step to make the build server automatically executing the following script

./scripts/build/dockerbuild.sh

This script will build the project, run quality assurance tests and push newly generated Docker image to the Docker Hub from which production servers will be able to pull the latest production changes as a part of the automated process. Image bellow shows example of Jenkins configuration for faerdapi-commit-stage

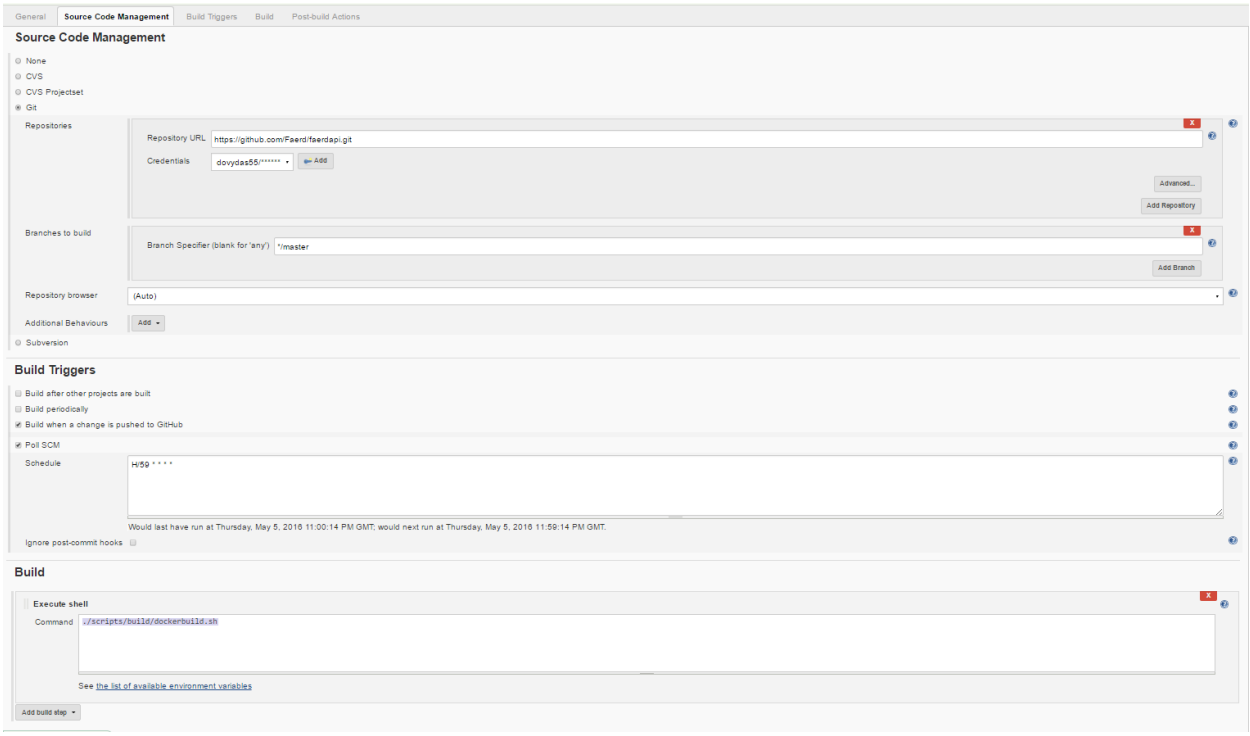


Figure 13: *displays faerdapi-commit-stage build configuration*

Do note that faerdapi-commit-stage does not push the latest changes from the VCS to the production. Instead, before pushing the latest changes acceptance tests and load tests have to be executed in order to make sure that new changes have not corrupted the build and functionality of API.
The next step is to create a new freestyle project called “faerdapi-acceptance-stage”. Once the project is created, select configure and add a build trigger to automatically build once

faerdapi-commit-stage has been built successfully. Finally add a build step to run acceptance tests by executing the following script

./scripts/test/run-acceptance-test.sh

Screenshot bellow shows the essential Jenkins configuration for faerdapi-acceptance-stage

Figure 14: *Jenkins configuration for faerdapi-acceptance-stage*

The next step is adding faerdapi-load-stage which will execute load tests against the API end points and makes sure that their performance passes the minimum expected standard. To do that, create a new project called “faerdapi-load-stage”. Once project is created select configure and add a build trigger to build once faerdapi-acceptance-stage has been built successfully. Finally add a build step to execute the load tests as follows

./scripts/test/run-load-tests.sh

Figure bellow shows the load stage configuration

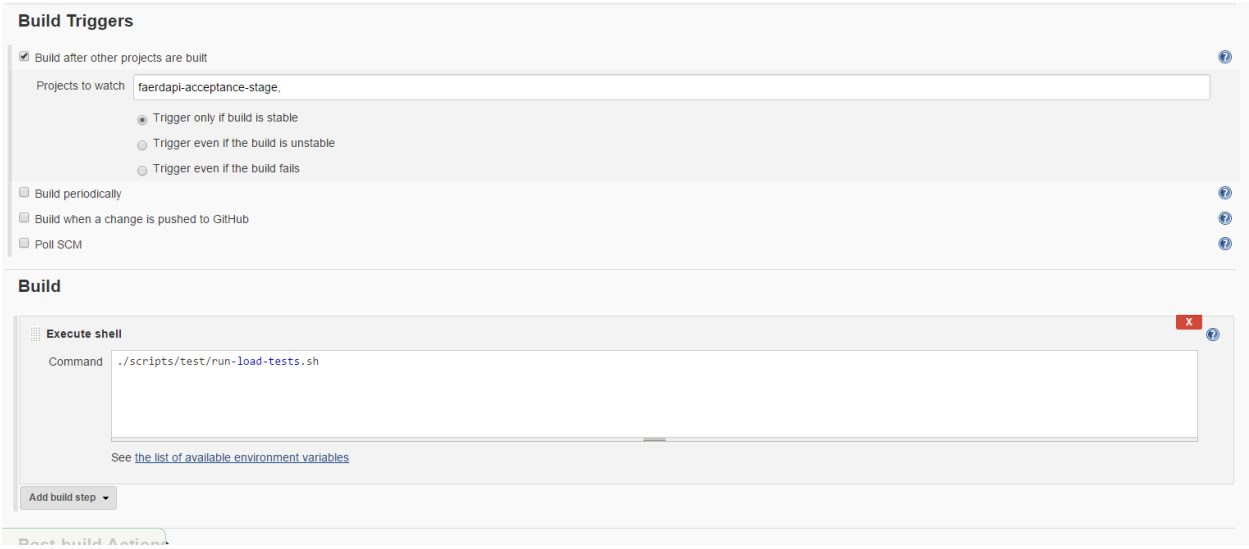


Figure 15: *shows build configuration for the load stage*

Once all of the tests have passes successfully it is time to push code changes to the production. This can be done by creating a new project called “faerdapi-production-stage”.

In this project add build trigger to execute after the load stage has built successfully. Finally add a build step to push latest code changes to the production by executing the following script

./scripts/build/synchronize.sh scripts/build/deployment.log

Figure bellow illustrates Jenkins configuration for the production stage

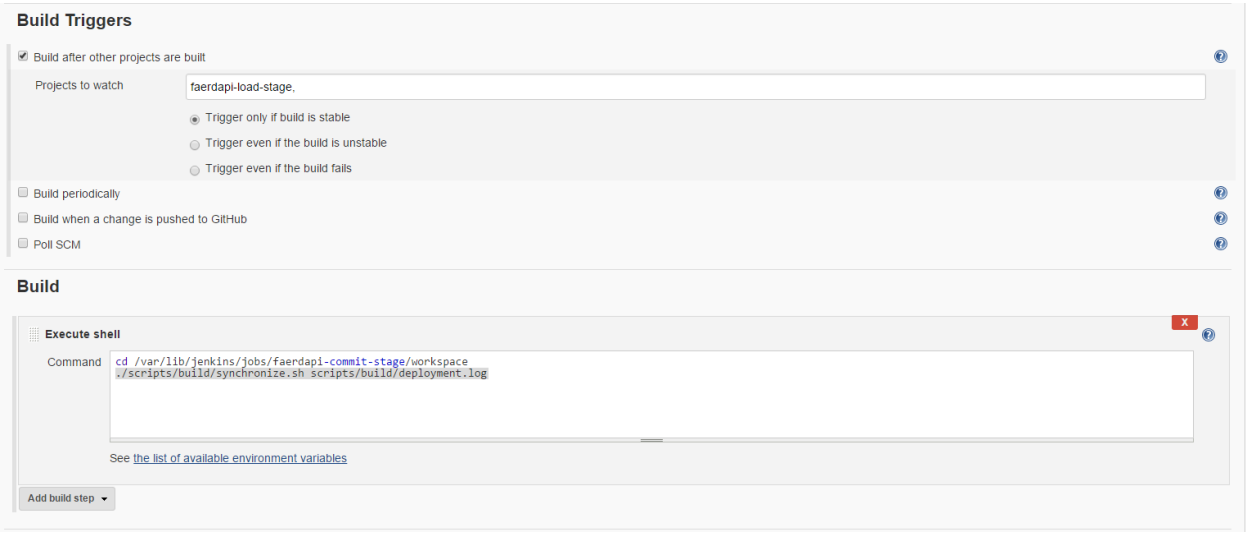


Figure 16: *illustrates configuration for the build stage*

Finally make sure to add build server's public key by appending it to the authorized keys inside each production machine. This will allow the build server to remotely execute commands in each production server and make continuous deployment possible. The ssh key can be cached by executing the following command

[~]~ ssh-copy-id username@productionServerIpAddress

Production server

In order to add brand new production machine to the list of production servers simply execute the following script to correctly install and configure all of the necessary dependencies needed to host the API.

[~faerdapi]# ./scripts/init/ubuntu-14.04-docker-init.sh

The next step is adding the server's IP address and PORT number on which the API is desired to be exposed to the deployment.log file which can be found under

scripts/build/deployment.log

The file has the following syntax: [USERNAME] [IP] [PORT]. Where username is the account name on which the server is going to be deployed, IP is the server IP address and PORT is the port number on which API is going to be accessible. Make sure to configure the firewall and open specified port so it would not block incoming traffic. Consider example of a deployment.log entry

dovydas 194.144.161.142 8000

It is not recommended to simply open a port through firewall and allow all of the incoming connections as it can be an obvious security vulnerability. Since all of the traffic is going to be forwarded by the load balancer, it is recommended to use intruder protection system (IPS) which would allow only connections forwarded by the load balancer and would scan for common attack vectors. As setup of such a system is very complicated procedure (could not be made a part of the automated process), the setup and configuration of such as software is the responsibly of the system administrator.
The final step for adding new production server is adding the IP address and PORT number of the new production machine to the Nginx load balancer which is typically dedicated machine (at least in our case) so it could efficiently handle high traffic. This can be achieved by editing nginx.conf file which can be found under

/etc/nginx/nginx.conf

By following these steps there is no limit how many machines can be added to the production and therefore this architecture is very easily scalable.

10. User manual

10.1 Downloading the app

The source code is attached to this report in a zipped folder. For instructions on how to install and run the project, please refer to chapter 9.1.

10.2 Using the app

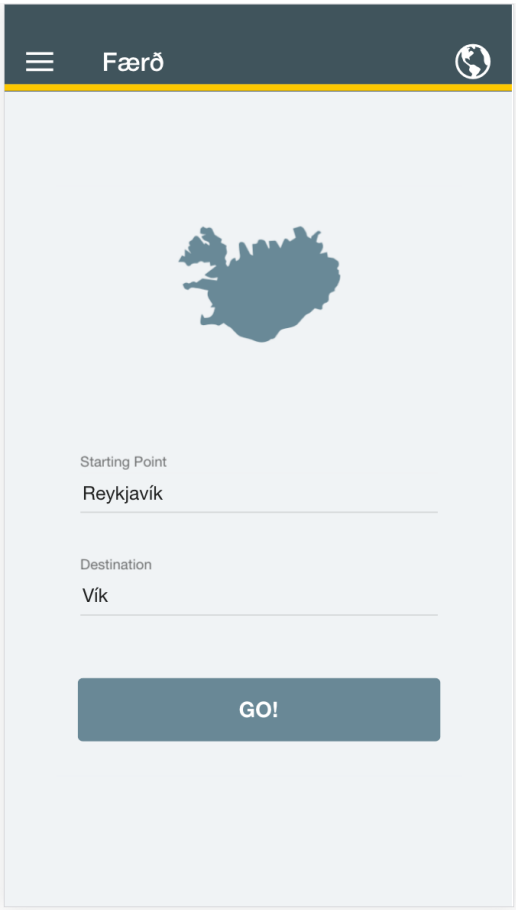


Figure 17: *Home screen*

a) Home screen

Opening the app brings up the home screen. The header at the top has two buttons that bring up menus. Tapping the globe on the right side opens the language and unit menu **(l)**. The menu on the left side brings up options that are not directly related to travel information **(h)**. The map at the center of the screen allows the user to explore road conditions everywhere in Iceland at once **(b)**, while the input fields below limit the output of information to a single route. By selecting two places from a list and pressing GO! only data regarding the shortest path between these two places will be displayed. On start the app fetches users current location but a diffrent start location can also be selected from a list.

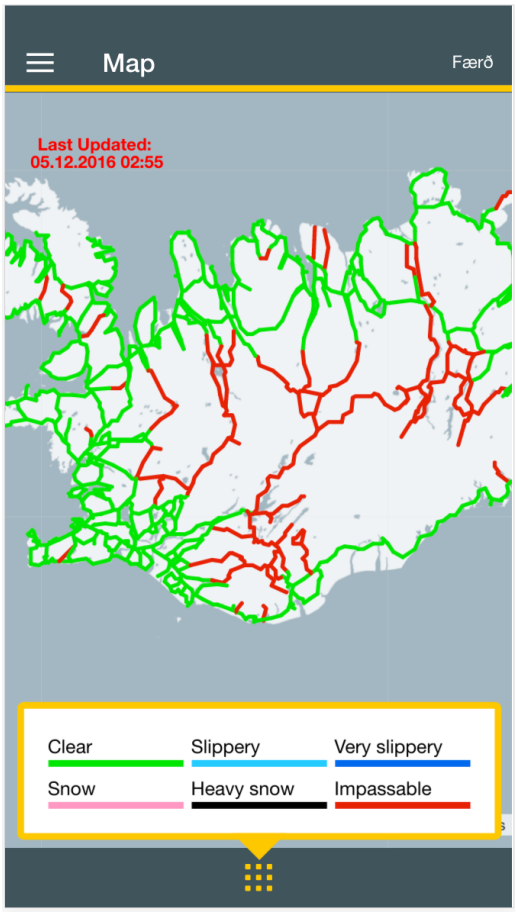


Figure 18: *Map – road conditions*

b) Road conditions
Conditions are displayed as color coded lines that represent the roads in Iceland. A key to the color codes is displayed in a banner at the bottom and can be collapsed by clicking the yellow button on the footer. The map can bee zoomed in and out to display a view over a the entire country or only a small area. Tapping the app name on the right brings the user back to the home screen **(a)**. The menu icon on the left of the header opens a menu for selecting different information to be displayed on the map **(c)**.

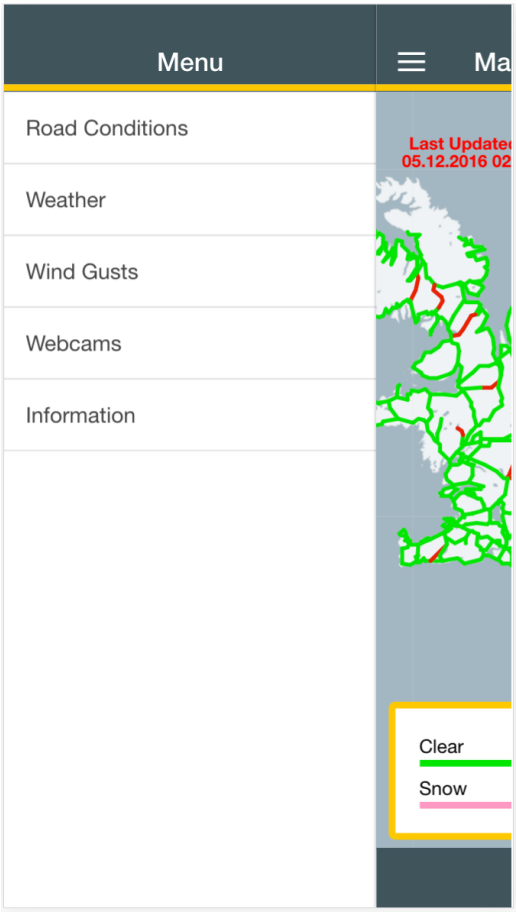


Figure 19: *Map - menu*

c) Map menu
The map menu allows the user to select what data is displayed on the map. By default the map opens with the road conditions on display, the other options are Weather **(d)**, Wind Gusts **(e)**, Webcams **(f)** and Information **(g)**.

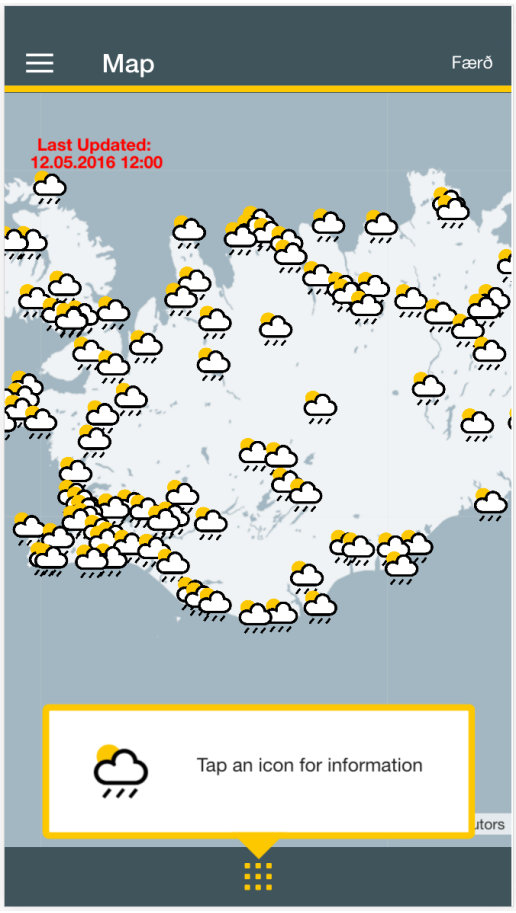


Figure 20: *Map - weather*

d) Weather

The map displays icons that represent the many weather stations that Vegagerðin operates around Iceland. Each icon can be tapped and the banner at the bottom of the screen will show the most recent data from that weather station, such as temperature and windspeed.

Figure 21: *Map – wind gusts*

e) Wind gusts

The map displays icons that represent windgusts that can potentially affect driving. They are categorized into four groups according to strength and the direction on the map accurately reflects the wind direction. Tapping the icons gives the user more detailed information about the exact strength of the gusts.

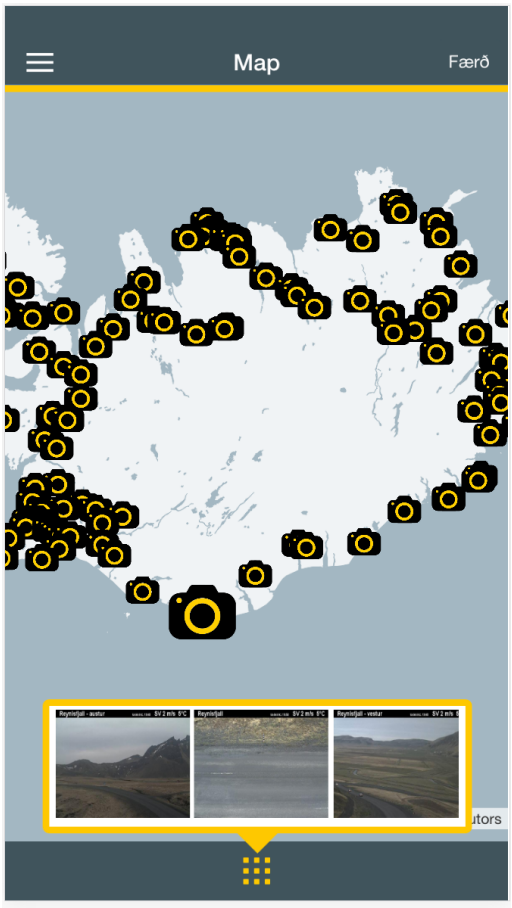


Figure 22: *Map - webcams*

f) Webcams

The camera icons each represent a group of web cameras that Vegagerðin operates in various locations around Iceland. Clicking one brings up thumbnails showing the latest photos from that location. The user can scroll horizontally through the thumbnails and enlarge them one at a time by tapping on them.

Figure 23: *Map - information*

g) Information

Road construction that can potentally slow down traffic is displayed as are restrictions on axisweight in certain areas. If the user has selected a single route, information concerning that route are displayed in the banner; distance, estimated fuel cost and time.

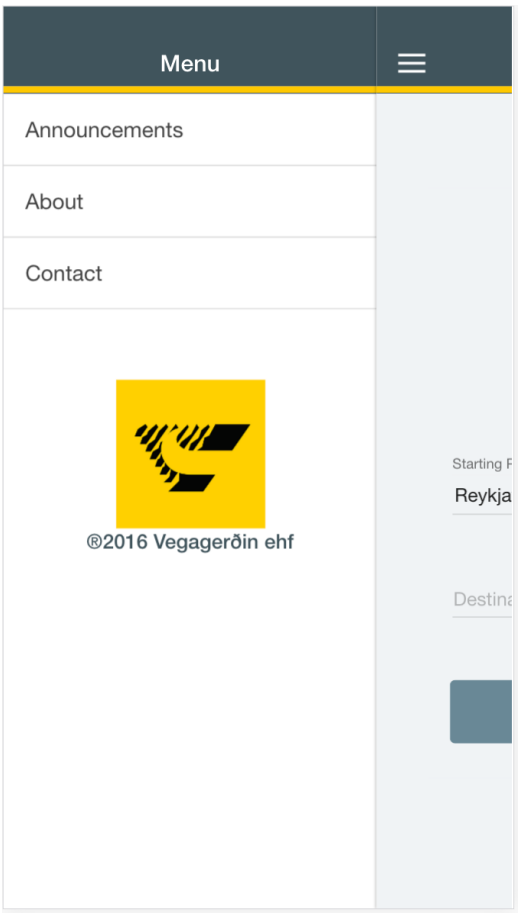


Figure 24: *Home – menu*

h) Side Menu

The menu to the side of the home screen **(a)** offers three options that are not directly linked to the road conditions displayed on the map; Announcements **(i)**, About **(j)** and Contact **(k)**. The Vegagerðin icon links to vegagerdin.is.

Figure 25: *Announcements*

i) Announcements

Important announcements regarding road conditions and safety are displayed. They are updated every day by Vegagerðin.

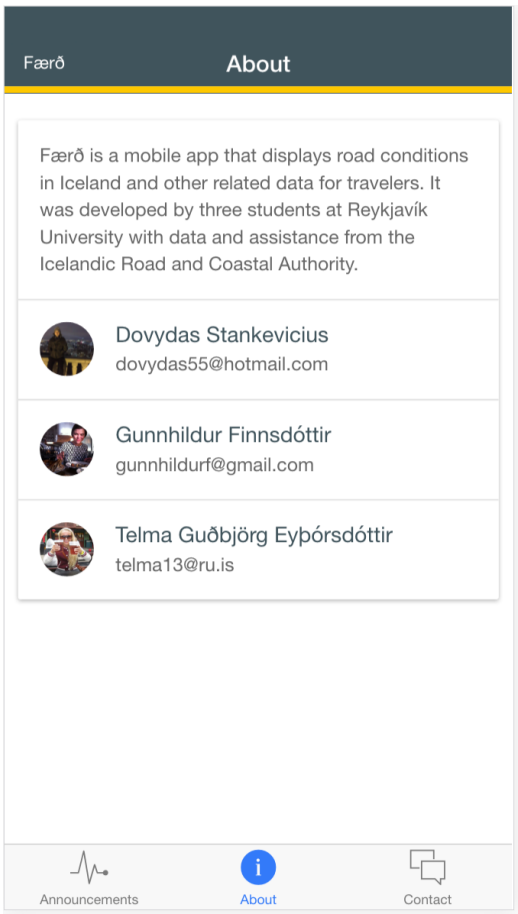


Figure 26: *About*

j) About

The About tab contains general information about the app, the data sources it uses and contact information for the development team.

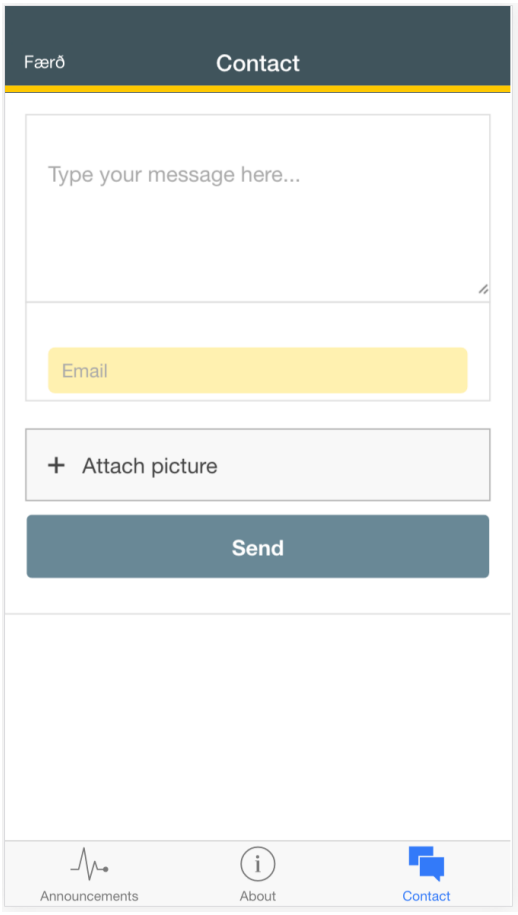


Figure 27: *Contact Vegagerðin*

k) Contact Vegagerðin

Færð offers users a contact form to send direct messages to Vegagerðin, for example if they notice damage in roads or signage. E-mail address is required, as is use of GPS location on the device. A message can be written in text and in addition the user can select a photo from library or take a new picture to further explain the issue in question.

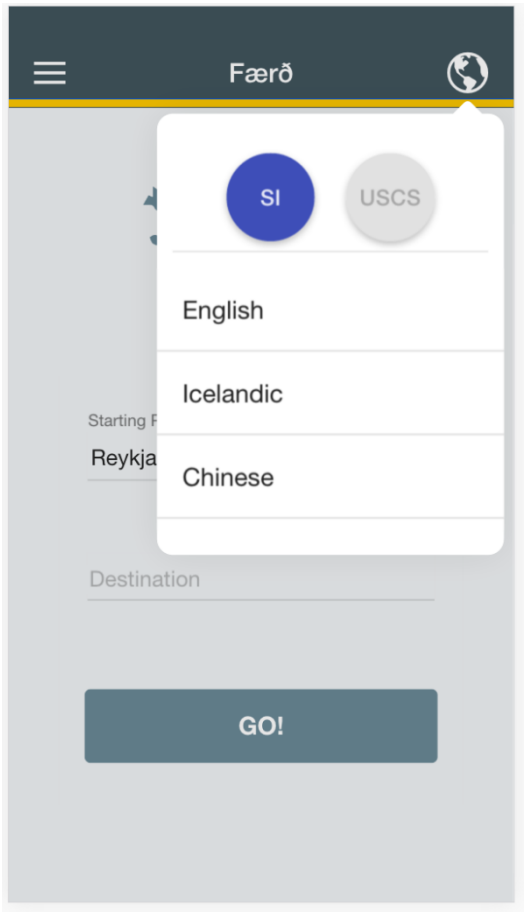


Figure 28: *Selection for language and units*

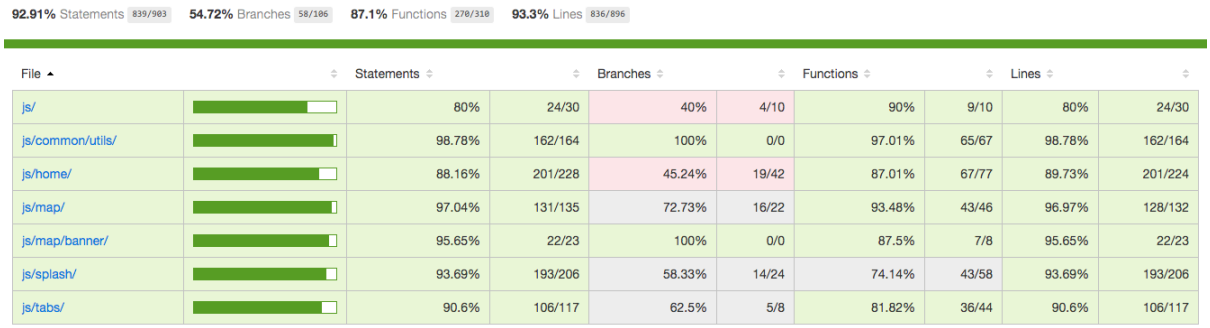
l) Languages and units

Færð supports a selection of languages. By default the app is set to English at download but by clicking the globe in the upper right corner of the home screen opens a menu where the user can choose from a list of ten languages. The same menu allows the user to select which units are displayed in the app, International System of Units (SI) or United Stated customary units (USCU). By keeping the former default option, the user will see information displayed in for example kilometers and liters where by switching to the latter it will appear in miles and gallons.

11. Conclusions

11.1 Results

At the end of this 15-week development cycle we have achieved all the main goals we set out within the beginning of this project. We have created a cross-platform app that runs on both iOS and Android with a variety of new client features such as language selection and access to weather data. We have ensured the quality of our source code by having over 90% overall code coverage and employing continuous deployment on the back-end and continuous integration on the client side. Our back-end is load balanced to improve efficiency and can easily be scaled up from the current two node clusters to handle heavy traffic once it is published. In addition, the analytic server can be used to both improve user experience and efficiency of the app.



92.91% Statements 839/903 **54.72%** Branches 58/106 **87.1%** Functions 270/310 **93.3%** Lines 836/896

File	Statements		Branches		Functions		Lines	
js/	80%	24/30	40%	4/10	90%	9/10	80%	24/30
js/common/utils/	98.78%	162/164	100%	0/0	97.01%	65/67	98.78%	162/164
js/home/	88.16%	201/228	45.24%	19/42	87.01%	67/77	89.73%	201/224
js/map/	97.04%	131/135	72.73%	16/22	93.48%	43/46	96.97%	128/132
js/map/banner/	95.65%	22/23	100%	0/0	87.5%	7/8	95.65%	22/23
js/splash/	93.69%	193/206	58.33%	14/24	74.14%	43/58	93.69%	193/206
js/tabs/	90.6%	106/117	62.5%	5/8	81.82%	36/44	90.6%	106/117

Figure 29: *90% code coverage*

11.2 Lessons learned

We attribute this success to a number of factors. First, using an established development methodology (Scrum) has enabled us to keep the project on track throughout the course of the past 15 weeks and have precise data on our progress at all times. Second, as we had previous experience with a similar project, we set out in the beginning with a very clear idea of our goals and a realistic plan to implement them. Third, our collaboration with Vegagerðin has been very beneficial for the project. Although all development was done by the team, having access to office space and advise from their technical staff has been very useful. And finally the three members of this team have all worked together on other projects before which made our teamwork very efficient.

As has been discussed in this report, we have had some twists and turns on our way towards the end result. We underestimated the workload in our other courses, especially around the end of the semester, and had to make some adjustments to our sprint planning accordingly. And although exploring different client development frameworks was a necessary part of the project, we could in hindsight have made our final choice sooner in the process. Most importantly, repeated attacks on our server have clearly demonstrated the importance of security in software development and we have implemented several safeguards to counter such attacks and minimize the damage they can cause.

11.3 Next steps

We have finished this chapter in the development of Færð but the project will continue to evolve. One team member has been hired by Vegagerðin to oversee further optimization and integration into their systems over the next months. Their specialists will no doubt have useful input on how to further improve the app and user testing by Vegagerðin staff around the country will provide insights into necessary adjustments. Publication is planned for the end of summer 2016 and we are confident that our work will result in a valuable service to those braving the icy roads of Iceland in the coming winter.