



Erhvervsakademi Sjælland

Afsluttende eksamensprojekt

revision, 5. semester

Prædefineret information

Startdato:	07-12-2018 12:00	Termin:	jan 2019
Slutdato:	07-01-2019 11:00	Bedømmelsesform:	Dansk 7-trinsskala
Eksamensform:	Mundtlig prøve	ECTS:	15
SIS-kode:	259410 0119 ro16da2a5-5H 19804 - MDT EKS 7TRIN		
Intern bedømmer:	Michael Claudius		

Deltager

Navn:	Jakub Kozak
Kandidatnr.:	8820 1335 jan 2019 609 3970
UNI-C ID:	(Ikke sat)
Alt. id:	0505962281
EASJ-id:	jaku0591@easj.dk

Gruppe

Gruppenavn:	Jakub Kozak
Gruppenummer:	24
Øvrige medlemmer:	Deltageren har afleveret i en enkeltmandsgruppe

MOLEHILL

social network connecting people nearby

Jakub Kozak



Zealand Institute
of Business and Technology

Supervisor: Michael Claudius

19/11/2018 – 06/01/2019

I hereby grant permission for this project report to be used by Erhvervsakademi Sjælland and permit the copying of this report as long as the content remains the same.

Signature:

A handwritten signature in blue ink, appearing to read 'Kozak', written in a cursive style.

Jakub Kozak

Date: 06/01/2019

Title: Molehill – social network connecting people nearby

Supervisor: Michael Claudius

Project period: 19th November 2018 – 6th January 2019

Keywords:

Web application, JavaScript, TypeScript, React, NodeJS, GraphQL, PostgreSQL, PostGIS, SCRUM, Extreme Programming

Abstract:

The purpose of this project is to develop a software application with focus mainly on the functionality. The development is accompanied by an agile software development process utilizing some of the practices from SCRUM and Extreme Programming. The final product of the project is a web application built with a help of JavaScript libraries, frameworks and PostgreSQL.

Table of Contents

1. Introduction	1
2. Problem Formulation.....	2
3. Methodology	3
3.1 Development Methodology	3
3.2 Scrum.....	3
3.2.1 Sprints.....	3
3.2.2 Product Owner	3
3.2.3 Sprint Planning.....	3
3.2.4 Daily Scrum	4
3.2.5 Sprint Review.....	4
3.2.6 Sprint Retrospective	4
3.2.7 Adjusted Scrum Practices	4
3.3 Extreme Programming.....	4
4. Technologies.....	6
4.1 Front-End.....	6
4.1.1 React.....	6
4.1.2 Redux.....	6
4.1.3 Miscellaneous.....	6
4.2 Back-End	7
4.2.1 Database Management System (DBMS)	7
4.2.2 Server.....	7
4.3 TypeScript.....	7
4.4 Architecture.....	8
5. Sprint 1	9
5.1 Introduction.....	9
5.2 User Stories.....	10
5.2.1 User Story 1	10
5.2.2 User Story 2	16
5.2.3 User Story 3	19
5.2.4 User Story 4	21
5.2.5 User Story 5	22
5.3 Sprint Review Meeting	22
5.4 Sprint Retrospective	23

6. Sprint 2	24
6.1 Introduction.....	24
6.2 User Story 6	24
6.3 User Story 7	25
6.4 User Story 8	26
6.5 User Story 9	26
6.6 User Story 10	27
6.7 Sprint Review Meeting	28
6.8 Sprint Retrospective	28
7. Sprint 3	29
7.1 Introduction.....	29
7.2 User Story 11	29
7.3 User Story 12	30
7.4 User Story 13	30
7.5 User Story 14	30
7.6 User Story 15	31
7.7 Sprint Review Meeting	31
7.8 Sprint Retrospective	31
8. Sprint 4	32
8.1 Introduction.....	32
8.2 User Story 16	32
8.4 User Story 18	33
8.3 User Story 17	34
8.5 User Story 19	34
8.6 Sprint Review Meeting	35
8.7 Sprint Retrospective	35
9. Sprint 5	37
9.1 Introduction.....	37
9.2 User Story 20	37
9.3 User Story 21	38
9.4 User Story 22	40
9.5 Sprint Review Meeting	41
9.6 Sprint Retrospective	41
10. Conclusion	42

10.1 Reflection.....	42
10.2 Evaluation.....	42
10.3 Next Steps.....	42
11. Bibliography / Webography.....	44

1. Introduction

This document serves as a final dissertation and is submitted as one of the requirements for the AP degree in Computer Science of Zealand Institute of Business and Technology, campus Roskilde. Its main aim is to showcase some of the fundamental skills acquired during the courses attended throughout four semesters, including the actual software implementation (coding) and use of an adequate software development methodology for a selected topic of interest.

In addition to that, the dissertation involves cooperation with a representative of the company I did an internship at. While working on the final project, I collaborated with Emil Lysgaard Hansen, a front-end engineer with rich experience in software development since 2011. He was my colleague in the Product Front-End Team at Famly during my internship period.

For this project I did not cooperate directly with the company (Famly) on the development of their product because they build one complex standalone application in teams; hence I would not have the opportunity to experiment with a broader domain of technologies and a development methodology on my own.

After discussing the dissertation requirements with Emil and other employees at Famly, he suggested I would use his application idea and utilize technologies we had used in the company to build upon what I had previously learned from them. The main difference in comparison with my internship is that this time I did not put hands only on the front-end part but on the back-end and an agile methodology as well.

Emil wanted a web application which allows users to share their current status and location with regards to what they are up to, so that all other users in the area can explore the statuses and respond to them. That was the core idea I was working on with a few adjustments and tweaks being refined during the whole development process.

2. Problem Formulation

In order to better understand the problem, scope and reasoning behind the dissertation topic addressed in this document, a few questions have to be stated first. It helps to stay focused on the relevant path during the application development by searching for answers in a well-defined domain.

How can I develop a social network using JavaScript technologies?

- What technologies are suitable for such project and why?
- How can the technologies be implemented?
- How to manage time and hand in the application on time?

If a social network wants to be successful, it needs many active users and that conveys potential problems with scalability and performance. For that reason, the right tools need to be used from the beginning to avoid unnecessary complications in the future.

Even the best tools are useless and can cause development nightmares without the right way of using them. I will attempt to follow the best recommended practices and deliver a performant app, so that users do not quit using the application only because long response (user interaction with UI) or load (fetching data) time.

Unfortunately, time assigned for this project is very limited, therefore I will have to decide what software development methodology is most efficient while managing a project with such strict time constraints and apply it accordingly.

3. Methodology

According to plan, the final result of this dissertation should be a functioning application ready to be tested by first users. I will investigate details of relevant technologies by reading their documentations, blog posts, forums and other available materials if necessary. After that I will implement the technologies in the required application.

3.1 Development Methodology

The project had to be well managed in order to be delivered by the given deadline. To increase the chance to achieve that, I had to choose and apply one of the development methodologies taught in the Computer Science programme.

I considered Waterfall development, Iterative and incremental development and Agile development as potential methodologies. Because Agile development is best designed for unpredictable changes, it looked like the most suitable approach.

I decided for an agile software development methodology mainly because I worked on the project alone and the lack of time; the development period lasted only 7 weeks. I had to take into account also some internal and external factors. E.g., in case of getting sick the project would come to a standstill for some time. For that reason, I had to be able to map progress and quickly adapt to changes by refining planned steps.

3.2 Scrum

I had previously worked in a team with an agile framework called Scrum during my internship. Even though it is designed for a development team with more than one member, I used some of its principles while developing the application.

3.2.1 Sprints

A sprint is a time-boxed incremental iteration in Scrum, which usually lasts between one week and one month. My sprint is one week long because I needed to deliver an increment, get a feedback and respond to it sufficiently often.

3.2.2 Product Owner

A Product Owner is a person whose responsibility includes making sure that the product value is maximized. He/she does that by creating a product backlog and prioritizing the backlog items. Apart from that he/she communicates with the development team and product's stakeholders, so it is clear what should be implemented and when.

In this case, my colleague, Emil Lysgaard Hansen, took the role of Product Owner, in the following chapters he is referred to just as `Product Owner`. We agreed on sharing the task of creating and ordering the backlog because that is also allowed if the Product Owner agrees with that.

3.2.3 Sprint Planning

A Sprint Planning is a meeting of the Scrum team (the development team, Product Owner) at which they choose items from the backlog for the upcoming Sprint. The Product Owner will give me advice or hints how to implement certain features if he knows the needed technology.

After each Sprint, I had a short informal meeting with the Product Owner. We discussed how the product development went during the last Sprint, went through the delivered working Increment while checking

against the user stories for the Sprint, and talked about features planned for the near future. Because we had time only one meeting per week, the Product Owner and I concluded that our Sprint Reviews will also have some signs of Sprint Planning.

3.2.4 Daily Scrum

In classical Scrum, members of the development team have to attend a short event every day, where each team member informs others about his work plan for the current day. The plan should bring the team closer to the Sprint goal. The members should also update the team on what they were working on the previous day and mention their concrete concerns regarding reaching the Sprint goal.

I will not have to talk to anybody like this during the Sprints, but I will go through very similar process every morning before I start coding or designing the application. I will make a to-do list for a day and order the tasks according to the priority level. Then the following day I will be also able to see what is left to do from the previous day and track the progress.

3.2.5 Sprint Review

A Sprint Review is a meeting held at the end of each Sprint where the result of the Sprint is discussed. The development team informs the Product Owner and stakeholders what was accomplished during the sprint and what they did not finish on time. The result of this meeting is a revised and potentially adjusted backlog.

3.2.6 Sprint Retrospective

A Sprint Retrospective is another Scrum meeting which takes place after the Sprint Review and prior to the next Sprint Planning. Its main purpose is to identify difficulties and unexpected problems experienced during the previous Sprint. The result of the meeting should be proposed Sprint improvements for the following Sprint.

In my case the Sprint Retrospective is not a typical one since I am the only team member. However, it can still be valuable time for self-reflection and reconsidering my efficiency with regards to the project.

3.2.7 Adjusted Scrum Practices

The Product Owner has a serious full-time job, so he could not spend too much time on this project. Therefore, we agreed to meet only once a week which will serve as both the Sprint Planning and Sprint Review.

A burndown chart is one of the Scrum artifacts used to monitor project progress. I will not use it because, in my opinion, it would not be of any help but unnecessary extra work. The length of each sprint will be only one week, and I am the only member of the Development team, so I assume, it will be easy to track the progress without it.

3.3 Extreme Programming

Extreme programming is a software development methodology often used in combination with Scrum. It is a set of practices and values designed to improve quality of software products by applying the best programming practices with an extreme approach. Compared to Scrum, it explicitly describes what rules programmers of the product should follow if they want to avoid problems with their code.

I decided to apply some of the practices of Extreme Programming for development of this application. Not all of them because some are applicable only for teams with multiple members. E.g., I cannot do pair

programming, even though I believe it can reduce code errors, because I am the only member of the development team.

Scrum is flexible with regards to the length of iterations. On the other hand, an Extreme Programming iteration must last only 1 week, which I find beneficial. It produces more frequent feedback and faster adjusting to customer needs.

All written code should have unit tests before the implemented feature is released. I will try to create unit tests for most of the code, but I expect they will slow me down since I do not possess much experience related to them. That can realistically result in abandoning full coverage unit testing in the attempt of finishing the application minimal viable product (MVP) on time.

Extreme Programming values simplicity over planning too much for the future iterations. I also try to create parts of the application only at the time when they are really needed. That should diminish time waste on avoidable features.

Nonetheless, sometimes there are circumstances when it is more constructive to refactor older code. It is not necessary but cleaner and more reusable code can save time for development in a long run. Refactoring is always worth considering but it can also cause premature optimization. That usually results in the opposite of expectations.

4. Technologies

Before the project setup can be established, the choice of technologies has to be clear. It would be much more time-consuming to replace a certain technology during the development after it was already implemented in certain parts of the application. Even though, technology replacement is sometimes unavoidable, good project setup can lower the chance of doing it in the future.

Some of the technologies, mainly those used on the front-end, were planned to be used from the beginning. I had good experience with them from previous projects and I did not find any better alternatives at the time of writing. I present more specific arguments when describing the individual technologies below.

4.1 Front-End

Because I am more experienced with front-end technologies, it was relatively easy to select the stack and tools. I knew that the social network had to be built as an application that could be used by a high number of people. I decided to develop it as a web application, because then it can be accessed from any web browser on possibly all devices. Web applications also remove the friction encountered when a native application has to be downloaded from Google Play Store or Apple App Store.

In addition to that, the web application can always be built with a tool, such as *Apache Cordova*¹ to get a mobile hybrid application running in Web views at a later point in time.

4.1.1 React

React is a popular front-end library for building composable and performant (if used correctly) single page applications (SPA). I chose React because of my personal experience with it and its massive and still growing popularity. Even though another front-end framework, *Vue.js*², passed React in the number of stars on GitHub, it is still the most used front-end library according to the number of *npm*³ downloads. It has a great support from the community and its creators from Facebook as well.

4.1.2 Redux

In comparison with other front-end frameworks (Vue.js, Angular), React is just a library which needs additional tools to be appropriate for applications with scope of higher complexity. Redux belongs to the category of such tools.

It is a library used for global state management not only in React applications. Another favourite library for state management is MobX but Redux passes it in both the number of GitHub stars and npm downloads.

4.1.3 Miscellaneous

As I already mentioned, React is not a fully-fledged framework on its own. It is meant to be used for reusable and composable visual components. For that reason, I used several libraries to fill in the React cavities.

- React-redux: connecting the Redux application state with React components
- React-dom: attaching React components to *DOM*
- React-router: mapping the current URL to a desired React component

¹ <https://cordova.apache.org/>

² <https://vuejs.org/>

³ <https://www.npmjs.com/>

- Redux-form: handling web forms for user input
- and a few more, described throughout the document

4.2 Back-End

The application needs to communicate with a server in order to connect users of the social network. The users, statuses and other data have to be also saved in a database to preserve them over time. The server and database could become a bottleneck of the application, so deciding what tools to use plays a vital role.

4.2.1 Database Management System (DBMS)

I considered multiple options while choosing the right DBMS, including a NoSQL document database, *MongoDB*⁴. A document database could cause several problems in a project where relationships occur naturally. Choosing the right efficient structure for database documents at the project setup phase is very difficult, if not nearly impossible, with agile development technology. The decision to correctly embed documents in other documents or reference them is not so straightforward if you adapt the plans and adjust the documents every week.

Therefore, I saw the type of relational database more applicable for the social network. I had to take into account another project requirement, saving the statuses together with location that can be displayed in a map on the front-end later.

That required a database supporting operations with *geographic information system*⁵ (GIS). The only free open-source DBMS satisfying the condition is PostgreSQL. It also allows to use GIS with a database extension called PostGIS.

4.2.2 Server

The front-end application is written with JavaScript libraries, so in order to minimize switching between language contexts, I selected *Express*⁶, a JavaScript web framework for *Node.js*⁷. This isomorphic application environment also potentially allows to add new full-stack developers for the project with more ease.

The Express server needs another library for being able to accept HTTP requests with *GraphQL*⁸ and process them accordingly. GraphQL is a query language from Facebook developed mainly for not over-fetching data. The biggest social network in the world uses GraphQL in production with more than 2 billion active users to make user experience more flawless in the parts of the world with not ideal internet connection.

4.3 TypeScript

I concluded that JavaScript, the language of the web, fits the project best. However, it is a dynamically typed language which can generate unexpected runtime errors. Not only is it more error prone than languages with static typing, but it is also more difficult for programmers to understand existing code with no types.

⁴ <https://www.mongodb.com/>

⁵ <https://gisgeography.com/what-gis-geographic-information-systems/>

⁶ <https://expressjs.com/>

⁷ <https://nodejs.org/en/>

⁸ <https://graphql.org/>

That can be solved by introducing *TypeScript*⁹ to the codebase, both on the front-end and back-end. It is a JavaScript superset with data types (number, string, boolean, object, array, function, enumeration, etc.) and interfaces which are compiled by the Typescript compiler into standard JavaScript code.

4.4 Architecture

The overall initial architecture can be seen in Fig.1. Users can access the web application built with React, Redux and other libraries, which will be mentioned in the later chapters, in a web browser. The web browser communicates with a server running Node.js environment. To handle user requests, I will use a Node.js framework Express together with Apollo Server. Apollo Server and Apollo Client are libraries for managing GraphQL requests and responses. All data that has to be persisted are saved in a PostgreSQL database. The database uses PostGIS extension for handling spatial data types and provides helper functions to manipulate these data types. In addition to that, code on both front-end and back-end will be written with TypeScript.

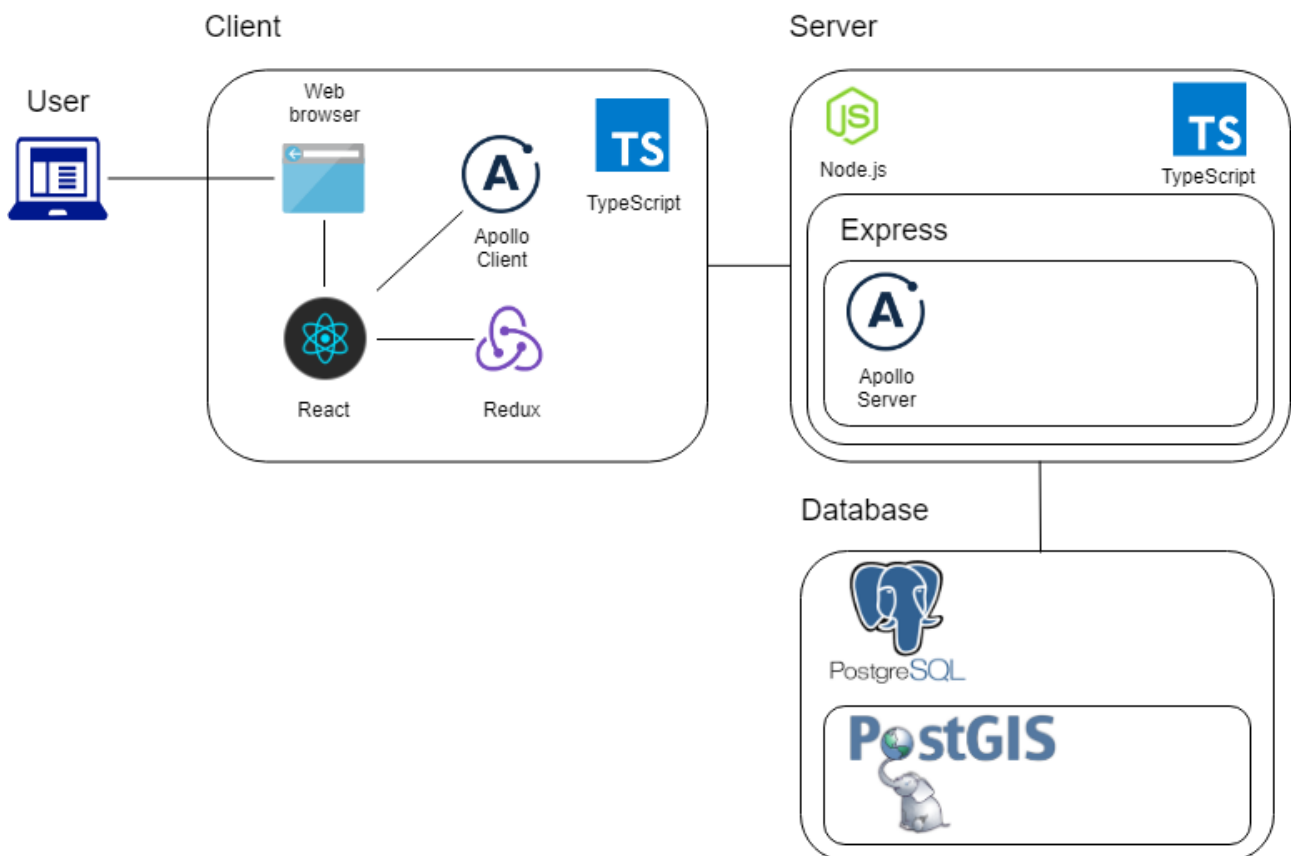


Fig. 1 Software architecture

⁹ <https://www.typescriptlang.org/>

5. Sprint 1

5.1 Introduction

After discussing the project with the Product Owner, I had a description good enough to start working on the required application. At our first meeting, he introduced his idea and we agreed on user stories that could be realistically finished by the project deadline.

The following user stories were formulated by the Product Owner and me as the Development Team. User stories are usually responsibility of the Product Owner, but other members of Scrum Team can also participate if the Product Owner decides to do so.

User stories – Entire project		Story points
1	Set up server environment (technical).	8
2	Set up web application environment (technical).	5
3	As a new user, I can sign up so that my account can be persisted over time on multiple devices.	5
4	As a member, I can log in so that I can see my account and have access to features only for members.	3
5	As a member, I can see a map with my location so that shown statuses are more relevant to me.	5
6	As a member, I can add a new status, so that other users can see what I find interesting.	8
7	As a member I can see a list of all statuses, so that I can take part in the community.	5
8	As a member, I can remove my statuses, so that they cannot be found anymore.	3
9	As a member, I can edit my statuses, so that I can correct mistakes.	5
10	As a member, I can filter statuses by radius, so that I can find statuses more relevant to my current location.	5
11	As member, I can comment statuses, so that other users know my opinion.	8
12	As a member, I can edit my comments, so that I can correct mistakes.	5
13	As a member, I can remove my comments, so that other users cannot see them anymore.	3
14	As a member, I can give infinite number of “likes” to statuses, so that I can mark what I find interesting.	3
15	As a member, I can see who gave “likes” to statuses and how many, so that I can decide how popular a certain status is.	5
16	As a member, I can have my user profile with a picture and a short description, so that other users can find out more about me.	8
17	As a member, I can see my statuses in my user profile, so that I have easier access to them.	5
18	As a member, I can filter statuses by category, so that I can find statuses more relevant to my interests.	8
19	As a member, I can use a map to select status location, so that I do not have to type the address manually.	5

20	As a member, I can mark a status with “join”, so that other users can see that I am coming to the status address.	8
21	As a member I can “join” a status privately, so that only the status creator can see that I am coming.	3
22	As a member, I can mark other users as friends/following, so that I can show my interest.	8
23	As a member, I can see notifications when people I follow, join a status, so that I can join the status with possibly mutual interests.	8

5.2 User Stories

The total number of story points is 129. Based on the sum and the time period for this project (5 sprints), I will assign the user stories to sprints. I should finish user stories with approximately 25.8 story points each sprint. Hence, I selected those 5 user stories (26 points) listed below for Sprint 1.

User stories – Sprint 1		
1	Set up server environment (technical).	
2	Set up web application environment (technical).	
3	As a new user, I can sign up so that my account can be persisted over time on multiple devices.	
4	As a member, I can log in so that I can see my account and have access to features only for members.	
5	As a member, I can see a map with my location so that shown statuses are more relevant to me.	

5.2.1 User Story 1

Before I could actually start coding, the project setup had to be prepared first. I decided to start with the backend part because it is a field I am less experienced with and it could take more time than anticipated. In my opinion, it is also easier to develop and test the frontend application if real data can be fetched from a running server.

5.2.1.1 PostgreSQL

For the beginning, the application will be running only in a local environment. Therefore, I had to download and install the PostgreSQL package on my Windows machine. The installation process was rather straightforward, I selected the folder for installed tools, port on which PostgreSQL will be running and kept all other default settings.

The package comes with *pgAdmin*¹⁰, which is a tool for database management and development. For instance, it can be used for viewing/changing a database structure, content and executing SQL queries against a selected database. I also used it to create the PostGIS extension for my database – molehilldb.

5.2.1.2 Server Setup

As I already mentioned, the server will run an Express application in Node.js environment. The server will not provide a standard, often used and still very popular REST API, but GraphQL will be used instead. GraphQL server expects to receive requests in a specific format and also sends responses in that format.

¹⁰ <https://www.pgadmin.org/>

I knew that I wanted to use TypeScript and GraphQL on both frontend and backend. It may be complicated to combine those two technologies because they both come with their own types. I had to make a little investigation on how to keep the types synchronized without writing certain type definitions twice.

I found a few useful libraries for this goal and installed them with npm, which is the package manager for JavaScript and the largest registry with all kinds of software packages. When a package is downloaded from npm, it is tracked with its version in the package.json file. In Fig.2, you can see how my package.json looked like after installing all needed packages.

```
1  {
2    "name": "typeorm-typegraphql",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "start": "ts-node -r tsconfig-paths/register src/bin/server.ts",
8      "build": "tsc -p .",
9      "dev": "nodemon .",
10     "configure-db": "cd ./src/bin && init-db.cmd"
11   },
12   "keywords": [],
13   "author": "Jakub Kozak",
14   "license": "ISC",
15   "dependencies": {
16     "apollo-server-express": "^2.1.0",
17     "express": "^4.16.3",
18     "geojson": "^0.5.0",
19     "nodemon": "^1.18.4",
20     "pg": "^7.5.0",
21     "reflect-metadata": "^0.1.12",
22     "ts-node": "^7.0.1",
23     "tsconfig-paths": "^3.6.0",
24     "type-graphql": "^0.14.0",
25     "typedi": "^0.8.0",
26     "typeorm": "^0.3.0-alpha.22",
27     "typeorm-typedi-extensions": "^0.2.1"
28   },
29   "devDependencies": {
30     "@types/geojson": "^7946.0.4",
31     "typescript": "^3.1.2"
32   }
33 }
```

Fig. 2 Package.json - Server Setup

The installed packages are listed under dependencies and devDependencies. Dependencies are such packages that must be bundled with rest of the code when used in production. On the other hand, devDependencies can be omitted in a production bundle because they are used only for development and testing purposes.

Now I will describe why the packages were installed and how they are used in the application.

apollo-server-express, express

The Express contains utility functions to handle HTTP requests. That can be used as the server core and to build API for data manipulation.

Apollo-server-express is a set of tools for implementing GraphQL on the server and can be used with Express. In Fig. 3, you can see how express and apollo-server-express are implemented.

```
1 import express, {Express} from 'express';
2 import {ApolloServer} from 'apollo-server-express';
3
4 import buildSchema from 'src/graphql/schema';
5
6 const startServer = (): Promise<{app: Express, server: ApolloServer}> => {
7   const app = express();
8
9   return new Promise((resolve) => {
10    buildSchema.then(schema => {
11      const server = new ApolloServer({schema});
12      server.applyMiddleware({ app });
13
14      resolve({app, server});
15    });
16  });
17 };
18
19 export default startServer;
```

Fig. 3 Server core

geojson, @types/geojson

In order to manipulate geometry data (status/user location), I will use geojson. @types/geojson contains TypeScript data types for geojson.

nodemon

The application will run locally, so I will use nodemon to run a local development server. One of the advantages of nodemon is that it watches code changes and restarts when some are found.

pg

A package that is used as PostgreSQL client for Node.js to handle connection to the database.

reflect-metadata

A library that adds the ability to use decorators with JavaScript classes. In this case, it is used only by *typeorm* and *type-graphql*.

ts-node

TypeScript Node allows to execute TypeScript files in Node.js.

tsconfig-paths

The way how TypeScript compiler works can be configured with the *tsconfig.json* file placed in the root directory. The file can contain *paths* object that maps keys to values when importing modules. The resultant path is relative to *baseUrl*. When TypeScript is used with Node.js, this package tells Node.js to check paths in *tsconfig.json* for module resolution.

```

1  {
2    "compilerOptions": {
3      "experimentalDecorators": true,
4      "target": "esnext",
5      "module": "commonjs",
6      "outDir": "build",
7      "sourceMap": true,
8      "noUnusedLocals": true,
9      "resolveJsonModule": true,
10     "allowSyntheticDefaultImports": true,
11     "esModuleInterop": true,
12     "moduleResolution": "node",
13     "emitDecoratorMetadata": true,
14     "baseUrl": ".",
15     "paths": {
16       "src/*": ["src/*"]
17     },
18     "lib": ["es2016", "esnext.asynciterable"]
19   },
20   "include": [
21     "**/*.ts",
22     "**/*.json"
23   ],
24   "exclude": [
25     "node_modules"
26   ]
27 }

```

Fig. 4 Server tsconfig.json

type-graphql

A library that allows to create GraphQL schema and resolvers with classes and decorators. GraphQL has its own language, Schema Definition Language (*SDL*) to write schemas. This package addresses the issue of having TypeScript types and SDL types.

typeorm, typeorm-typedi-extensions

Packages that are used for object-relation mapping (ORM) and work also with PostgreSQL.

When all packages were in place, I created a database table for users by defining an entity with typeorm.

Note that the described code and screenshots in this section are from User stories 3 and 4.

A class becomes an entity if it is decorated with `@Entity`. Typeorm adds a column to a table if a class property has the `@Column` decorator. Each column can have further specified properties/constraints with an object passed to `@Column` as an argument. In Fig 5, User has an id that is an autogenerated and auto increased primary key. The User table also contains two columns with unique values (username, email) of type varchar. Password column is also of type varchar but

```

1  import {Field, ID, ObjectType} from 'type-graphql';
2  import {Entity, PrimaryGeneratedColumn, Column} from 'typeorm';
3
4  You, a few seconds ago | 1 author (You)
5  @Entity()
6  @ObjectType()
7  export default class User {
8    @Field(() => ID)
9    @PrimaryGeneratedColumn()
10   id: string;
11
12   @Field()
13   @Column({unique: true})
14   username: string;
15
16   @Field()
17   @Column({unique: true})
18   email: string;
19
20   @Field()
21   @Column()

```

Fig. 5 User entity

not unique. When the server starts, and the User table does not exist, typeorm connects to the database and executes the query shown in Fig.6.

```
creating a new table: user
query: CREATE TABLE "user" ("id" SERIAL NOT NULL, "username" character varying NOT NULL, "email" character varying NOT NULL, "password" character varying NOT NULL, CONSTRAINT "UQ_78a916df40e02a9deb1c4b75e db" UNIQUE ("username"), CONSTRAINT "UQ_e12875dfb3b1d92d7d7c5377e22" UNIQUE ("email"), CONSTRAINT "PK_cace4a159ff9f2512dd42373760" PRIMARY KEY ("id"))
```

Fig. 6 Creating the User table

In the Fig. 5, you can see two more decorators - @ObjectType and @Field. The first one tells TypeScript to create a GraphQL type and the latter adds the decorated properties to the generated type. Because GraphQL types include String and ID, the @Field decorator needs an argument that decides in case of User id, when the type is ambiguous.

Fig.7 shows the generated GraphQL type for the class User.

Then I used the User entity for basic business logic, i.e. create user, find user and log in user. With GraphQL, this is done with resolvers. A resolver is a function that can take arguments sent from the client and is called when its corresponding schema path is hit. Its result is then sent back to the client.

```
type User {
  id: ID!
  username: String!
  email: String!
  password: String!
}
```

Fig. 7 Generated User type

GraphQL uses two types of requests for handling data – Query and Mutation. Query is in essence used with GET requests for fetching data. Mutation, as its name suggests, mutates existing data. In this case, creating a new user is a mutation because it creates new data.

Fig. 8 is a resolver that handles data related to User. A class becomes a resolver if it is decorated with @Resolver from type-graphql. The methods are added to the GraphQL schema as Queries and Mutations if they have corresponding decorators.

In the constructor of UserResolver I injected userRepository, which is an object used to access and process data in the database. On line 22, a user is found by id, on lines 27-30 a user is found by provided email and password while logging in and on line 37 a new

```
1 import {
2   Resolver,
3   Query,
4   Mutation,
5   Arg,
6 } from 'type-graphql';
7 import { Repository } from 'typeorm';
8 import { InjectRepository } from 'typeorm-typedi-extensions';
9
10 import UserEntity from 'src/entity/user';
11 import { SignUpInput, LoginInput } from 'src/graphql/resolvers/user/input';
12
13 @Resolver(of => UserEntity)
14 export default class UserResolver {
15
16   constructor(
17     @InjectRepository(UserEntity) private readonly userRepository: Repository<UserEntity>,
18   ) {}
19
20   @Query((returns) => UserEntity)
21   async user(@Arg('id') userId: string): Promise<UserEntity> {
22     return await this.userRepository.findOne({
23       id: userId,
24     });
25
26   @Mutation((returns) => UserEntity)
27   async login(@Arg('loginInput') loginInput: LoginInput): Promise<UserEntity | undefined> {
28     const foundUser = await this.userRepository.findOne({
29       email: loginInput.email,
30       password: loginInput.password,
31     });
32     return foundUser;
33   }
34
35   @Mutation(returns => UserEntity)
36   async createUser(@Arg('user') newUser: SignUpInput): Promise<Partial<UserEntity> | any> {
37     const savedUser = await this.userRepository.save(newUser);
38     return savedUser;
39   }
40 }
```

Fig. 8 User resolver

user is created and saved in the database. The only argument of the methods is data from the client. If the argument is not a primitive data type, it has to be another special decorated class as shown in Fig. 9.

Without these classes (SignUpInput, LoginInput) decorated with *@InputType*, type-graphql would not be able to correctly infer GraphQL types.

The complete generated GraphQL schema is shown in Fig. 10 (User story 3-4). *type Query* shows that a user can be fetched by id and *type Mutation* has two available mutations with arguments LoginInput and SignUpInput. All queries and mutations return data of type User.

Before I started with the front-end application, I needed to confirm that the prepared API worked as expected. Apollo-server-express comes with a built-in development tool for this purpose. It can be accessed on the same domain and port as the express server and route */graphql*. In Fig.11, I tested the createUser mutation with query variables in the bottom left corner of the screen. On the right part of the screen is the response from the server. It contains all requested data including the autogenerated user id. I tested also the login mutation and user query and they all worked as intended. That was a proof that the backend for this sprint is done and the next step was the frontend application.

```
1 import {InputType, Field} from 'type-graphql';
2
3 import UserEntity from 'src/entity/user';
4
5 @InputType()
6 export class SignUpInput implements Partial<UserEntity> {
7   @Field()
8   username: string;
9
10  @Field()
11  email: string;
12
13  @Field()
14  password: string;
15 }
16
17 @InputType()
18 export class LoginInput implements Partial<UserEntity> {
19   @Field()
20   email: string;
21
22   @Field()
23   password: string;
24 }
```

Fig. 9 User input types

```
input LoginInput {
  email: String!
  password: String!
}

type Mutation {
  login(loginInput: LoginInput!): User!
  createUser(user: SignUpInput!): User!
}

type Query {
  user(id: String!): User!
}

input SignUpInput {
  username: String!
  email: String!
  password: String!
}

type User {
  id: ID!
  username: String!
  email: String!
  password: String!
}
```

Fig. 10 GraphQL schema - User

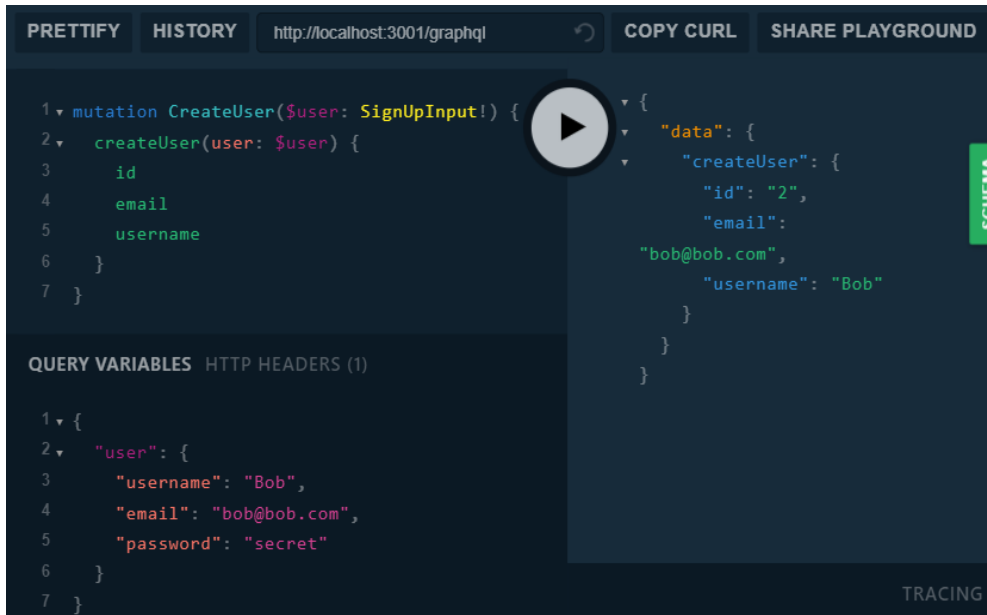


Fig. 9 GraphQL IDE - GraphiQL

5.2.2 User Story 2

The beginning of the web application was very similar to the server. I made a little research on what packages would be needed and installed them. They can be found in Fig. 12.

apollo-boost

- creates the apollo client for communication with the server GraphQL API

connected-react-router

- used to keep routing data in the application state

graphql

- used internally by apollo-boost

history

- creates an instance of browser history used by connected-react-history

immutable

- provides immutable data structures to prevent React from unnecessary re-rendering

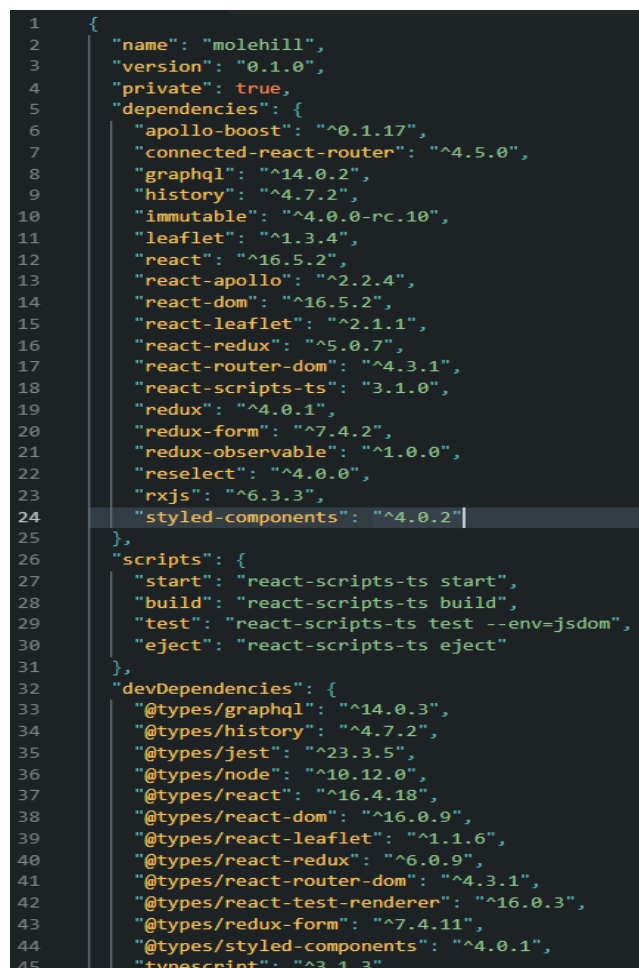


Fig. 10 Package.json - Web application setup

leaflet

- displays maps and provides the way to manipulate them

react

- core library of the application
- used for creating visual UI components

react-apollo

- allows to use apollo-client in the context of React

react-dom

- serves as bridge between the React virtual DOM and actual HTML DOM
- renders React components in HTML DOM

react-leaflet

- leaflet maps as React components

react-redux

- allows to use Redux application state in the context of React

react-router-dom

- router management for web applications

react-scripts-ts

- scripts that bootstrap a React application with TypeScript with no initial build configuration

redux

- makes global application state management easier

redux-form

- manages application state for html forms in Redux

redux-observable

- helps with handling asynchronous actions and side effects of Redux actions

reselect

- creates selector functions for deriving state from the Redux state
- caches input and output of the selector functions and prevents unnecessary re-rendering of React components

rxjs

- reactive extensions library for JavaScript using Observables

styled-components

- better way of styling React components with no css classes

devDependencies

- TypeScript types for the aforementioned packages
- TypeScript itself for compiling TS files to JavaScript executable by any popular web browser

With the packages installed, the project needed UI for user sign-up, login and an overview with a map showing the current user location. The UI had to be able to send GraphQL requests and handle corresponding responses.

Fig. 13 shows the main App component with the whole application after I finished the sprint. Each pair of tags is a React component. *ErrorBoundary* is a custom component that handles errors thrown in any child component. *Provider*, *ApolloProvider* and *ThemeProvider* are third-party components from the installed packages. *Provider* passes down Redux store that can be accessed by all child components and contains the global application state and a method that can mutate the state. *ApolloProvider* accepts an apollo client for handling GraphQL requests and keeps locally cached data fetched from the GraphQL server. *ThemeProvider* allows to access the styling *theme* object in any child component, so that the application look can be easily changed from a single file. *GlobalCSS* overrides default browser CSS rules and sets some custom rules applied to all HTML elements. *Portal* displays its child component in an HTML element with a given id. Its goal is to render the child outside the main element with the React application so it can be rendered anywhere. *GlobalEvent* shows a notification if something interesting happens, e.g. something was (un)successfully saved. *Routes* holds a component with routing logic.

```
class App extends Component {
  public render() {
    return (
      <ErrorBoundary>
        <Provider store={store}>
          <ApolloProvider client={client}>
            <ThemeProvider theme={theme}>
              <>
                <GlobalCSS theme={theme} />
                <Portal id="global-event">
                  <GlobalEvent />
                </Portal>
                <Routes />
              </>
            </ThemeProvider>
          </ApolloProvider>
        </Provider>
      </ErrorBoundary>
    );
  }
}
```

Fig. 11 Main App component

The Product Owner's requests needed three routes – Login, SignUp and Overview. The outermost component *ConnectedRouter* is in charge of mapping the browser URL to a component and render it. *Switch* is another third-party component used for grouping multiple routes. *PrivateRoute* is a custom component that wraps *Route* and either renders the component

```
const Routes: React.SFC = () => (
  <ConnectedRouter history={history}>
    <Switch>
      <Route exact path="/" component={Login}/>
      <Route path="/signup" component={SignUp}/>
      <PrivateRoute path="/overview" component={Overview} />
    </Switch>
  </ConnectedRouter>
);
```

Fig. 12 Routes - Sprint 1

passed down as a prop, if user is logged in, or redirects the user to the Login page. The component checks if user is saved in the local storage to decide.

5.2.3 User Story 3

The SignUp component in Fig. 15 is wrapped with a higher order components (HOC) in the export statement. A HOC is a component that receives a child component with all its props, adds more props and passes them down to the child component.

The Sign-up form has four input fields – email, username, password, passwordRepeat and a submit button. If the mutation is successful and a new user is created, *loginSuccess* action is dispatched, which adds the user data to the local storage and Redux state as well.

I decided to go more into details of the Form because it uses another HOC, more specifically *reduxForm* from the *redux-form* package. The only mandatory property of the configuration object in the *reduxForm* argument is *form*, which is its name. It is needed to keep its internal state in Redux state. I also used *validate* to confirm a few conditions before its values are sent to the server and avoid unnecessary processing of invalid data. *Validate* in the *reduxForm* HOC is used for global form validation. Apart from that, you can see a *validate* prop in *FormField-s* for email and username. They are used for local validation of single input fields. The validation functions (Fig. 16) receive the current value of their input fields and must return *undefined* if no problem is found. If something other than *undefined* is returned, it can be then accessed in the corresponding fields to notify user about what is wrong.

The last thing worth noticing in the form is *Button*. It is another custom component that is stylable and looks same throughout the whole application.

With the Login and SignUp, it is easy to see how efficient it can be to work with reusable generic components.

Rendered SignUp is shown in Fig 18.

```
const SignUp = (props: Props) => {  
  
  const handleSubmit = (values: ISignUpFormData) => {  
    const {passwordRepeat, ...newUser} = values;  
  
    props.mutate({  
      variables: {  
        user: newUser,  
      },  
    }).then((response) => {  
      if (response) {  
        loginSuccess.dispatch(response.data.createUser);  
      }  
    }).catch(e => e);  
  };  
  
  return (  
    <ScreenCenter>  
      <Headline textAlign="center">Molehill</Headline>  
      <Base marginTop={s2}>  
        <Body textAlign="center">  
          | Sign up to see what people around you are up to  
        </Body>  
      </Base>  
      <Base marginTop={s4}>  
        <Form onSubmit={handleSubmit} />  
      </Base>  
      <Body marginTop={s4} textAlign="center">  
        | Do you have an account already?  
        <Link to="/">Log in</Link>  
      </Body>  
    </ScreenCenter>  
  );  
};  
  
export default compose(  
  graphql<<{}>, Response, ISignUpFormData>(signupMutation),  
)<SignUp>;
```

Fig. 13 SignUp component

```
export const email = (value: string) =>  
  value && /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$/i.test(value) ?  
  'Invalid email address' : undefined;  
  
export const maxLength = (max: number) => (value: string) =>  
  value && value.length > max ? `Must be ${max} characters or less` : undefined;  
  
export const minLength = (min: number) => (value: string) =>  
  value && value.length < min ? `Must be ${min} characters or more` : undefined;
```

Fig. 14 SignUp - form validation

```

const minLength3 = minLength(3);
const maxLength24 = maxLength(24);

const SignUpForm: React.SFC<Props> = props => {
  const { handleSubmit } = props;

  return (
    <Form onSubmit={handleSubmit}>
      <FormField
        required
        validate={email}
        type="email"
        name="email"
        component={FormInput}
        placeholder="Email address"
      />
      <FormField
        required
        validate={[minLength3, maxLength24]}
        name="username"
        component={FormInput}
        placeholder="Username"
      />
      <FormField
        required
        name="password"
        component={FormInput}
        type="password"
        placeholder="Password"
      />
      <FormField
        required
        name="passwordRepeat"
        component={FormInput}
        type="password"
        placeholder="Repeat password"
      />
      <Button
        text="Sign up"
        type="submit"
        appearance="submit"
        fullWidth={true}
      />
    </Form>
  );
};

const validate = (values: ISignUpFormData) => {
  const errors: Partial<ISignUpFormData> = {};

  if (values.password !== values.passwordRepeat) {
    errors.passwordRepeat = `Those passwords didn't match. Try again.`;
  }

  return errors;
};

export default compose(
  reduxForm({
    form: 'SIGNUP_FORM',
    validate,
  }),
)(SignUpForm);

```

Fig. 17 SignUp - Form

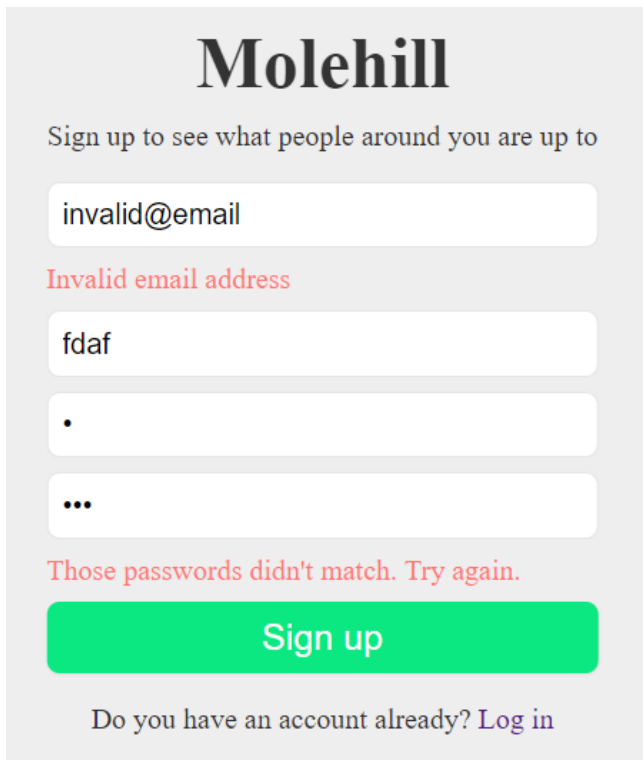


Fig. 18 Sign-up page

5.2.3.1 GraphQL connection

Fig. 18 depicts an example of a standard process of handling user input and communication between GraphQL Client and Server. Mutations and Queries are handled the same way, only request payloads differ.

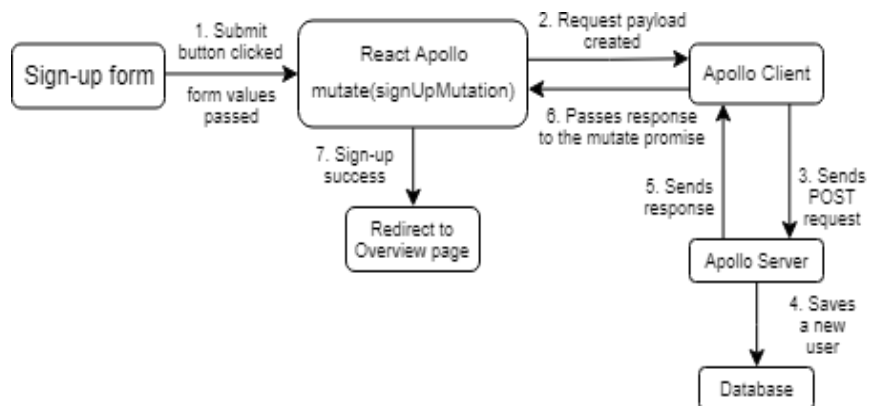


Fig. 18 Sign-up process - diagram

5.2.3.2 Redux in action

Fig. 19 shows how the global Redux state is updated by dispatching an action. A Redux action is just an object that must contain a property with a key *type*. And a reducer is a function that takes the global state and an action as arguments, checks if the action with a certain type should update the state. If the state updates, React runs its reconciliation algorithm to check if any properties of currently rendered components have been changed. If so, React re-renders the updated component, so UI contains the latest information.

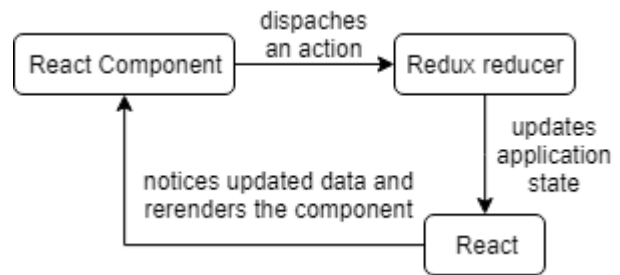


Fig. 19 React component lifecycle

5.2.4 User Story 4

The Login component (Fig.20) is made of many components used also in SignUp. The changes can be seen in the export statement, where the graphql HOC receives a different mutation and even though the Form component (Fig. 20) has the same name, it refers to a different imported component. The graphql HOC gets *loginMutation* (Fig. 21) and creates a *mutate* function which can enable the GraphQL client to send mutation to the server. *withStateMutation* is a custom HOC that tracks loading state of mutations.

The Login's content is fairly simple. All components except *Link* are custom. *ScreenCenter* is used only for styling purposes, it positions its content in the center of current view. *Headline* is a text component with certain style properties and is aligned in the center with CSS. *Base* is just a standard div element whose margins and paddings can be styled with properties passed down from outside. *Form* contains two input fields for user email and password and a button for submitting the form. It receives the *loading* prop so the form button can show a spinner if it is true. By clicking the button, user calls the login mutation that is received by the server, the form values are checked against users in the database and the user is redirected to the Overview page on success. Otherwise user is notified by GlobalEvent that their username/password combination is invalid. *Body* is another text component that can be styled with *props* as well as *Headline*. *Link* is a component of react-router-dom that is a usual HTML anchor tag but it just changes route, instead of redirecting and full page reload.

```
const Login: React.SFC<Props> = (props) => {
  const {
    sMutation,
  } = props;

  const onSubmit = (values: FormData) => {
    return sMutation.mutate({
      variables: {
        loginInput: values,
      },
    }).then(response => {
      if (response) {
        loginSuccess.dispatch(response.data.login);
      }
    }).catch(e => updateError.dispatch(
      'Invalid username/password combination.',
    ));
  };

  return (
    <ScreenCenter>
      <Headline textAlign="center">Molehill</Headline>
      <Base marginTop={s4}>
        <Form loading={sMutation.loading} onSubmit={onSubmit}/>
      </Base>
      <Body marginTop={s4} textAlign="center">
        Don't have an account?
        <Link to="/signup">Sign up</Link>
      </Body>
    </ScreenCenter>
  );
};

export default compose(
  graphql(loginMutation),
  withStateMutation(),
)(Login);
```

Fig. 20 Login component

```
import gql from 'graphql-tag';

export const loginMutation = gql`
  mutation login($loginInput: LoginInput!) {
    login(loginInput: $loginInput) {
      id
      username
      email
    }
  }
`
```

Fig. 21 Login mutation

The result rendered by the Login component is shown in Fig. 22.

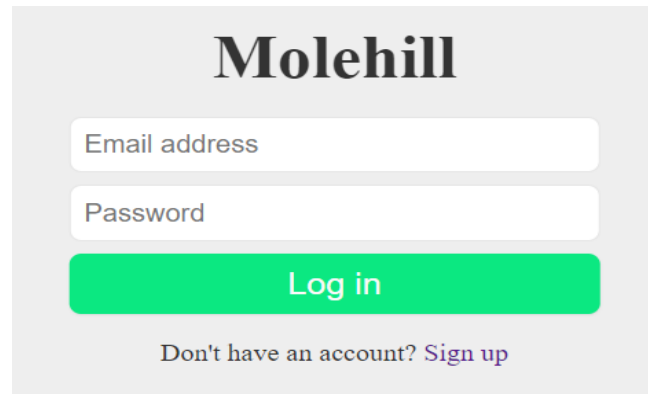


Fig. 22 Login page

5.2.5 User Story 5

The only new thing worth mentioning in *Overview* is the process of getting user location (Fig. 23). Because *Overview* is a class component, `getLocation` method is created in its constructor. Then it is called in `componentDidMount`¹¹ lifecycle hook which is called only once, when a component is rendered for the first time. Another useful hook - `componentWillUnmount`, called before the component is destroyed, is used to clear location watcher.

`getLocation` first checks if a browser supports geolocation API. If it does, user location is saved in the application state and is being watched for changes. After that, longitude and latitude are supplied to the map from `react-leaflet` to show a pin (Fig. 24). I also added a button that is supposed to add a status in the given location. In my opinion, it will help the Product Owner visualize next steps.

```
this.getLocation = () => {
  if (navigator.geolocation) {
    return navigator.geolocation.watchPosition(
      this.geolocationSuccessCallback,
      this.geolocationErrorCallback,
      {enableHighAccuracy: true},
    );
  }
  else {
    this.isGeolocationAvailable = false;
    return 0;
  }
};
this.getLocation = this.getLocation.bind(this);
}

public componentDidMount() {
  this.watchId = this.getLocation();
}

public componentWillUnmount() {
  if (this.watchId) {
    navigator.geolocation.clearwatch(this.watchId);
  }
}
```

Fig. 23 Getting user location

With Login, Sign-Up and the map with user location, goals of all three user stories have been accomplished.

5.3 Sprint Review Meeting

At the meeting I demonstrated how users can create an account, use it to log in and how it is followed by redirecting to the map with their location. The Product Owner was also interested in the project setup, from the perspective of an experienced software developer. He wanted to see how the system blocks are connected in order to give me advice in case something could be done better. That would not only help me

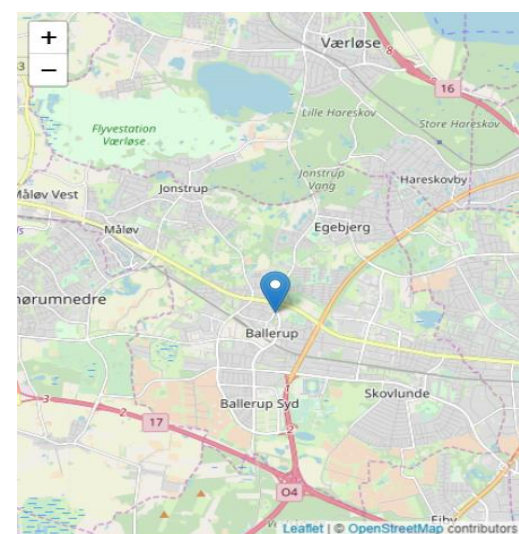


Fig. 24 User location in the map

¹¹ <https://reactjs.org/docs/react-component.html>

with development but speed the entire project up. He could not find a better way, so we agreed that the project foundation is solid.

We talked about my following Sprint plans and he sketched a rough design how the application could look after the second Sprint. We discussed that and I took his suggestions into account.

5.4 Sprint Retrospective

Even though there was nobody else in the Development Team except me, the Sprint Retrospective could identify potentially harmful activities and technologies for the project. At the end of each Sprint, I will make 2 lists of observations – “Went well”, so I can see what brings value and I should continue using, and “Could be improved” so I know what causes problems, slows me down and I should reconsider.

Went well

- TypeScript caught many errors before compiling the code
- TypeScript improved intellisense in my code editor
- GraphQL¹² is a great tool to test and improve GraphQL queries and mutations before implementing them in the web application
- Realistically estimated user stories were an important aspect for good time management
- Morning overview of plans for the day by creating a simple to-do list of tasks

Could be improved

- Some npm packages have wrong TypeScript types in the latest, not well user tested, releases
 - I should investigate that next time I need to install a new package
- When I need to use a GraphQL type from the schema, I need to look for it in GraphQL or the backend code
 - I should autogenerate TypeScript types from the GraphQL schema types.

¹² <https://github.com/graphql/graphiql>

6. Sprint 2

6.1 Introduction

The last Sprint Review Meeting with the Product Owner played also a role of the Sprint Planning. We selected user stories for the following Sprint whose sum of story points is close to 25.8, the average of story points for each planned Sprint. Another factor influencing the choice of user stories was the overall significance of the application feature accomplished by the selected user stories.

The most vital part of the Molehill application is statuses, therefore that is the first feature, which was implemented in the section for users after they log in.

User stories – Sprint 2		Story points
6	As a member, I can add a new status, so that other users can see what I find interesting.	8
7	As a member I can see a list of all statuses, so that I can take part in the community.	5
8	As a member, I can remove my statuses, so that they cannot be found anymore.	3
9	As a member, I can edit my statuses, so that I can correct mistakes.	5
10	As a member, I can filter statuses by radius, so that I can find statuses more relevant to my current location.	5

6.2 User Story 6

Because the whole idea around Molehill is about statuses, I had to add a way to create some. In addition to that, all other user stories in this Sprint needed at least one existing status created by a user.

I started with creating a Status entity with type-orm that creates a database table with all necessary columns. The entity also provides an access to the table and can perform all operations available in a PostgreSQL database – save a new status, update and delete an existing status, find statuses and their relations with given conditions.

The next step was to create a GraphQL resolver that contains a mutation for adding a new status. I had to define a GraphQL type for the acceptable structure of a new status and add business logic that handles incoming statuses. If a new status wants to use current location, longitude and latitude, the resolver uses a third-party geocoder API from OpenStreetMap to get its address. Once the address is complete, the status is saved in the database and its result is sent back to the client.

I tested this backend part with the GraphQL (Fig. 11) and pgAdmin. When its correctness was proved, I moved to implementing UI in the web application. I created a Redux action that is dispatched when the 'Add' button is clicked. That action opens a modal with a form containing custom styled form fields.

Users can manually fill out all the form fields – title, description, country, city, postcode, street number. In Fig. 25, you can see a radio button 'Use current location' that sends a query with current longitude and latitude to the server endpoint. The server then uses the OpenStreetMap API for reversed geocoding and the UI for address is automatically filled out with the response data.

When the form submit button with text '+ Add' is clicked, a corresponding mutation is sent with the form values and the status is saved.

6.3 User Story 7

The main application page, the one that users see immediately after they log in, must hold a feed with statuses. The statuses should be ordered by time they were created, and users should see only those whose location is not out of 30-kilometre radius.

Again, I started with preparing a backend API that can load statuses with given criteria from the database. I already had a few statuses saved, so the API could be tested with GraphQL and pgAdmin.

Then I created a query in the web application, that would fetch statuses for the feed. I placed the query in a React component that would hold a list of all retrieved statuses. I reviewed the structure of received data and built another React component for a single status (Fig. 26). Because each status comes with coordinates, I used it to build a URL that navigates from the current location to the status place with Google Maps.

The loaded statuses are also displayed in the map (Fig. 27). The picture of a mole is a user and the blue pins around mark statuses nearby. This was done with react-leaflet npm package. When a status in the map is selected, the corresponding status is marked as active in the feed and vice versa.

Fig. 25 Add status - Form

Fig. 26 Status item

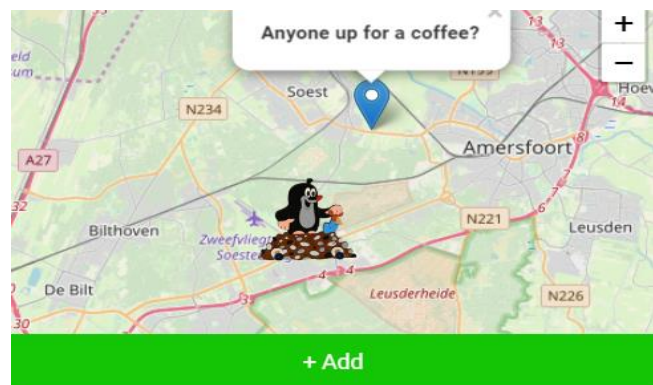


Fig. 27 Status in the map

6.4 User Story 8

The initial process for this user story copied the previous one. Before any new UI components were added, I wrote a delete mutation and tested it.

I created generic React components for a dropdown that can take any number of options and a dialog window that can be used for asking any permission from a user. I added an option 'Delete' that opens the dialog window when clicked (Fig. 28). The dialog contains two buttons – one to confirm an intended decision and one to cancel the current activity in progress (Fig. 29). When the status deletion is approved also in the dialog, a GraphQL mutation removes the status for good.

6.5 User Story 9

Users also need a simple way to correct their mistakes in created statuses. It can be a simple typo in a title or description, or a wrongly inserted address. That could mislead users who would potentially want to join the posted status.

A simple SQL query can do that with a known id and updated content of a status. Because only fetched statuses can be edited, I had all necessary information ready on the front-end. I added a new option 'Edit' to the dropdown from the User Story 8 (Fig. 30). This time it does not open a dialog window when clicked but a modal with a form (Fig 31). The form has the same input fields as the form for adding a status. The form fields are then populated with initial values from the existing status being edited.

The mechanism of Apollo Client cache can automatically update the edited status in the rendered map and the status feed. The only condition is that the GraphQL server must send a response that contains a GraphQL type 'Status', id of the status and updated fields.

However, the Apollo Client does not know how to update cached data when adding or removing a new item with mutations. Therefore, I had to manually access the cache and update the statuses when removing and adding them in previous user stories.

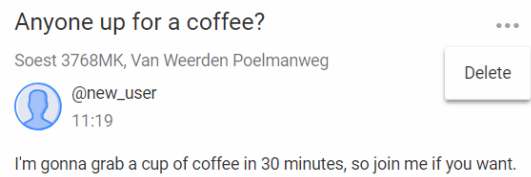


Fig. 28 Menu button – delete

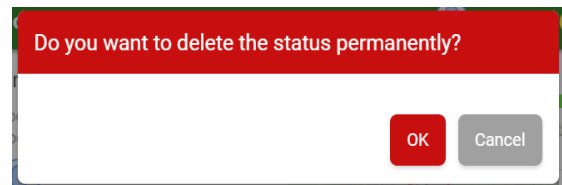


Fig. 29 Delete status confirmation

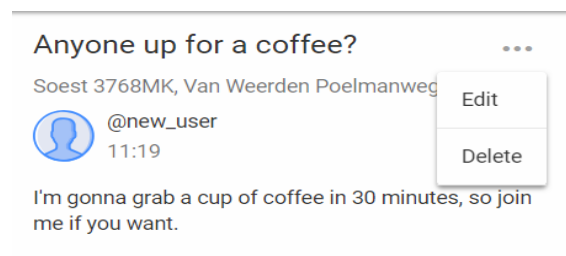


Fig. 30 Menu button - edit

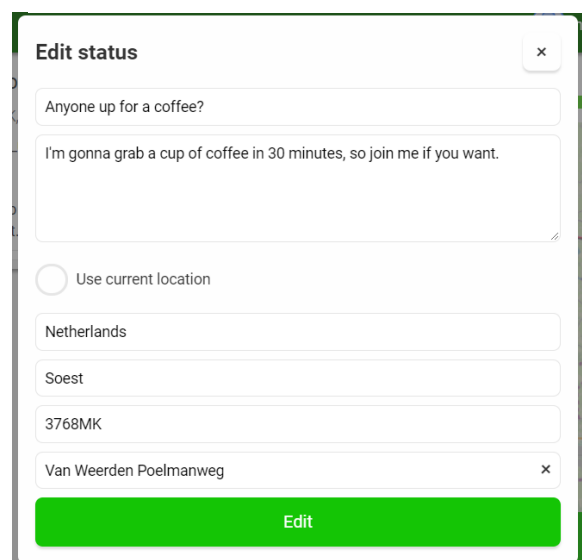


Fig. 31 Edit status -form

6.6 User Story 10

Statuses in the map and feed have to be always relevant to the current user location. Until this point, all of them were from radius of 30 kilometres around the user. It can be a too wide range in case a user wants to get to the posted location only by foot. Therefore, fetched statuses had to be filtered by a given radius.

To get statuses in a given radius I needed to get the radius from user first. I used another npm package – rc-slider, that can be given a range of values, a step by which a selected value can be shifted and exposes the selected value.

In this user story, I utilized two of special helper functions provided for PostgreSQL by the PostGIS extension for the first time. That function can easily use and compare geospatial data, in this case Point of latitude and longitude (Fig. 33). In the where clause, you can see how the function `ST_Distance_Sphere` calculates the distance in meters between two points. The `ST_MakePoint` function creates a new geometry point with latitude and longitude from a user. So, the where clause compares the distance between a user and a status location and selects only those statuses where the calculated distance is less than or equal to the radius selected by user.

Another interesting and very useful technique used for this user story involved `redux-observable`. Normally the Apollo Client sends a request every time when at least one of the query parameters is changed. That would mean that statuses would be fetched always after the radius is changed. Even though the Apollo cache would limit that only to 30 requests – the radius range is 30 kilometres and one slider step is 1 kilometre, it would still be a very inefficient solution. And that is when `redux-observable` can save the day.

When the radius is changed in the application state by dispatching a Redux action, it also pauses the automatic system of fetching statuses (Fig. 34). A `redux-observable Epic` in charge of that action resumes the fetching only if the same action has not been dispatched for 900ms. That means user can move the slider handle as much as he wants, but a request is sent only if he stops for 900ms.

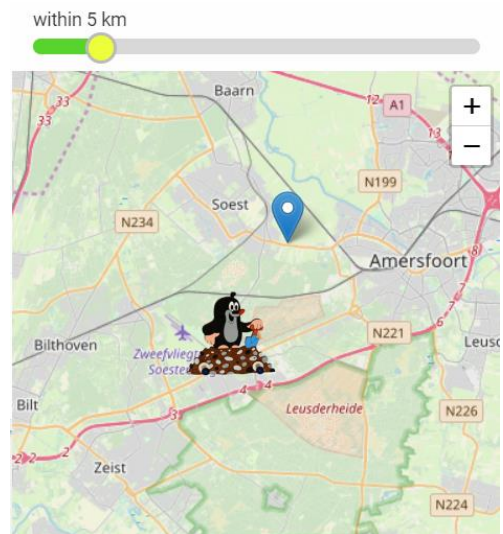


Fig. 32 Search within radius

```
const statusesWithUser = await ctx.statusRepository
  .createQueryBuilder('status')
  .leftJoinAndSelect('status.user', 'user')
  .where('ST_Distance_Sphere(location, ST_MakePoint(:latitude,:longitude)) <= :radius', {
    radius,
    latitude,
    longitude,
  })
  .andWhere(cursor ? `status.id < ${cursor}` : 'TRUE')
  .take(limit ? limit : BIG_INT_LIMIT)
  .orderBy('status.createdAt', 'DESC')
  .getManyAndCount();
```

Fig. 33 Geospatial query

```
const fetchStatusesOnRadiusChange: Epic<IReduxAction, any> = action$ =>
  action$.pipe(
    ofType(changeRadius.type),
    debounceTime(900),
    mergeMap(() => {
      return of(startAutoRefetchStatuses.action());
    }),
  );
```

Fig. 34 Request filtering - code

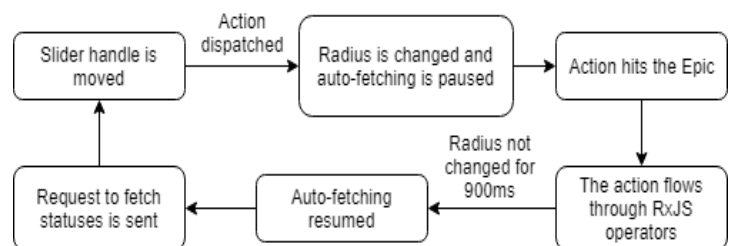


Fig. 35 Request filtering - diagram

6.7 Sprint Review Meeting

The process how we managed the meeting had nothing new in comparison with the previous Sprint Review Meeting. Only the Product Owner and I took part in the review.

First, I showed him new features finished during the Sprint. He liked that their design was very close to what we agreed on beforehand. The Product Owner was also satisfied with its technical functionality. I explained him how we prevent the application from over-fetching and re-fetching by using GraphQL, Redux Observable and optimistic updates with the Apollo Client. All those technologies combined significantly improve user experience by decreasing response time.

After we tested the new features, we discussed future development. That included mainly user stories for the upcoming Sprint and their details. Again, he sketched very brief design of alternative visual components, but the final decision was up to me.

6.8 Sprint Retrospective

Went well

- Collaboration with the Product Owner was very straightforward
- It was motivational to see the fundamental part of the application in place
- PostGIS has a robust official documentation, so it was not complicated to write queries with its functions

Could be improved

- Redux-form library can be a little tricky because it requires a specific syntax and puts complexity into a black box
 - I should read its entire documentation and find some examples
- Third-party libraries can stop you from reinventing the wheel. On the other hand, they require you to use their syntax, specific order, etc. For that reason, I spent quite a lot of time resolving issues with Redux Observable.
 - I should be careful when introducing new libraries to the codebase. I will have to consider how much time it can save and waste in a long run.

7. Sprint 3

7.1 Introduction

At this point, a user could create an account and post a new status visible to other users. However, there is still no way how users can interact with each other. Without any user interaction, I assume, the probability of users coming back would decrease.

Therefore, I will implement very common features for social network statuses/posts. Users should be able to comment all visible statuses, so they feel like they are not alone but a part of bigger community. The second feature, that will be accomplished with user stories below, is the ability to express with “likes” which statuses are more interesting.

User stories – Sprint 3		Story points
11	As member, I can comment statuses, so that other users know my opinion.	8
12	As a member, I can edit my comments, so that I can correct mistakes.	5
13	As a member, I can remove my comments, so that other users cannot see them anymore.	3
14	As a member, I can give infinite number of “likes” to statuses, so that I can mark what I find interesting.	3
15	As a member, I can see who gave “likes” to statuses and how many, so that I can decide how popular a certain status is.	5

7.2 User Story 11

Until now, I got used to a certain development workflow. I always started with the backend, where I created a type-orm entity. The entity added a new database table with specified columns. With that, I had a way how to persist all status comments.

It was followed by a new GraphQL resolver for comments. The first step was to have a mutation that saves an incoming comment in the corresponding database table and returns at least a generated id of the new comment.

Once the mutation was tested with dummy data in GraphQL, I could add a user interface to the front-end. Because users like to see what they are used to, I decided to just add a simple input field at the bottom of status item and a button (Fig. 36).

A new comment is shown below the status when the green “Comment” button is clicked. It is done by updating data in the Apollo Client cache, so no unnecessary request is used.

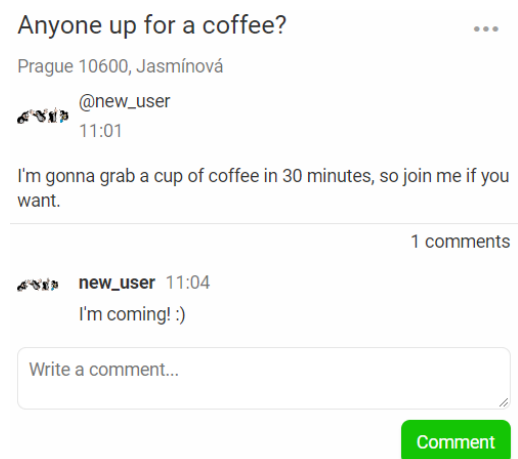


Fig. 36 Status comment

A status can have an infinite number of comments, therefore I also added pagination. Initially, only three latest comments are loaded and by clicking “Load more” five more comments are fetched.

7.3 User Story 12

It would be unfortunate if a user had to delete a comment and retype it with corrections for a single typo. Another case is when a user wants to update information in his comment.

The backend part needs only a new mutation in the resolver. The mutation must be able to receive a text of updated comment and comment id, so the record can be changed.

The Redux state keeps data if a comment is in edit-mode. If it is (Fig. 38), the comment body is changed to textarea and by pressing Enter-key, change of the comment is confirmed.

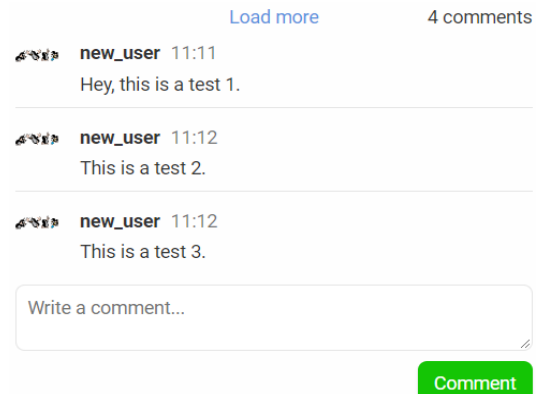


Fig. 37 Status comments - pagination

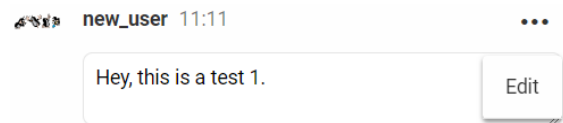


Fig. 38 Status comment - edit mode

7.4 User Story 13

User story 8 (status deletion) was almost identical to this one. I needed only comment id which was later used in a GraphQL mutation to delete a certain comment. The dropdown button with “Edit” and “Delete” options was a generic React component, so I had to provide only a button text and what should happen when clicked.

When the “Delete” option is clicked, a generic dialog window with a custom message is open (Fig. 39). It is the same procedure as in the User story 8. A user can then confirm comment deletion and a list of comments is updated.

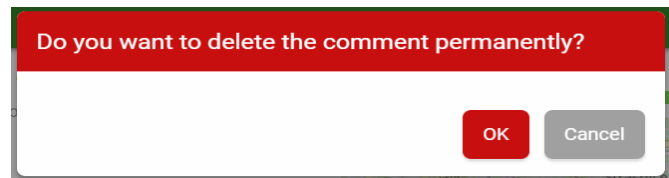


Fig. 39 Delete comment confirmation

7.5 User Story 14

The Product Owner requested a system of likes to have some data to measure popularity of statuses among users. Usually it is possible to give only one like or dislike to a post/status/article. However, I was asked to try to do something different, the Product Owner wanted unlimited number of likes for each user. The plan is to come up with an optimal algorithm to take that into account in the future when filtering statuses by popularity.

I added an icon below the status body (Fig. 40), when it is clicked, a new like is stored with relations to a user and status. The total number of likes for a status is shown next to the “thumbs-up” icon. For now, users have to decide on their own which statuses are currently popular. It should be automatized for convenience with some filters in the future.

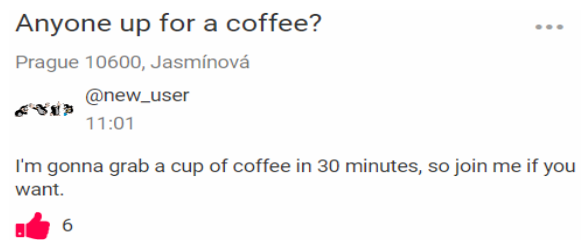


Fig. 40 Status likes

7.6 User Story 15

It is always important to see who gave likes to a status. If a user, who is going through a list of statuses in his area, is somehow related to another user who likes a certain status, then that status can potentially be a more relevant than others. In Molehill, each user can give an infinite number of likes, so that makes it even more important to see who and how many likes gave to a status.

Fig. 41 shows what users see when the number next to the thumbs-up icon is clicked. In the image you can see an open modal window and that a user with username "new_user" pressed the icon seven times. It provides enough information for users to decide whether the status is popular or not.

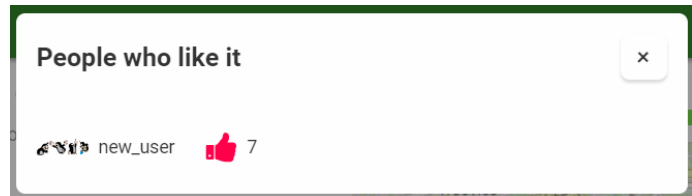


Fig. 41 Status likes - overview

7.7 Sprint Review Meeting

The Increment after accomplishing the user stories for this Sprint certainly extended the application according to the requirements from the Product Owner. He was again satisfied with the delivered work and could see great progress from the time the project had been started.

He liked that I thought about pagination for status comments and implemented it right away. He was also glad to see that some visual React components are completely generic and can be reused for multiple purposes. That should save some time always when we need to add a new feature.

We discussed already planned user stories and also future improvements of the existing features, such as likes for comments, comments for comments, then the previously mentioned algorithm for determining status popularity, etc. We focused mainly on the following Sprint and the improvements were only something we could think about in the meantime.

7.8 Sprint Retrospective

Went well

- Previously made generic components were a great help
- I got more familiar with third-party libraries used in Molehill
- Simple minimalistic design looks good and is easy to create without deeper understanding of design principles

Could be improved

- The generic visual components could be refactored, and their performance could be tested later if they are used on multiples places throughout the application

8. Sprint 4

8.1 Introduction

The user stories of previous Sprint (comments, likes) have been done in order to support a better user interaction, so that the platform feels to be more active. The user stories 16-17 will continue in this manner, they will help users make their accounts more transparent to others. The other user stories, 18 and 19, will try to make application usage easier by extension of previously finished features.

User stories – Sprint 4		Story points
16	As a member, I can have my user profile with a picture and a short description, so that other users can find out more about me.	8
17	As a member, I can see my statuses in my user profile, so that I have easier access to them.	5
18	As a member, I can filter statuses by category, so that I can find statuses more relevant to my interests.	8
19	As a member, I can use a map to select status location, so that I do not have to type the address manually.	5

8.2 User Story 16

Until now, all users had accounts with very basic information, such as their email address, username, password and all of them had a default image. This could cause natural complications with user trustworthiness. Without any personal information visible to other users, they would not look like real people. Even though, adding a picture and a short user description is still not a bulletproof evidence, it might increase trust among users.

I extended a User database table with two columns – image and bio. The image column holds a location path to a saved image. The bio column is a text entered by a user with any information, he considers important and interesting for users who will visit his profile in the application.

The user bio is relatively easy to implement, I used the generic modal window that holds a Redux form with only one field – textarea for the bio itself (Fig. 42). And I placed it in a new route for a user profile.

In order to add a user image, I had to install two new packages, one on the front-end (apollo-upload-client) and the other on the back-end (graphql-upload). In essence, they allow to send and parse files in GraphQL queries and mutations.

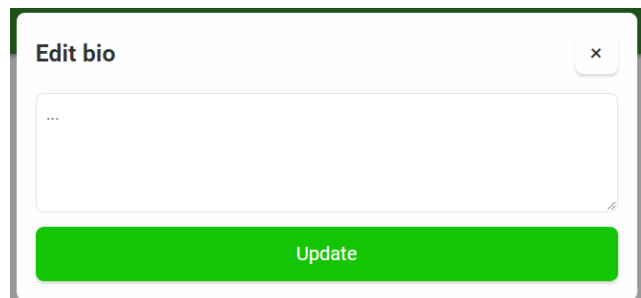


Fig. 42 User bio

Fig. 43 shows the editable user profile. When the “New profile image” is clicked, file explorer opens, and a user can select an image from the file system.

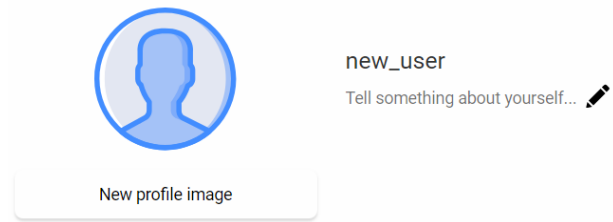


Fig. 43 User profile

Fig. 44 depicts how the uploaded image is handled by a mutation in the User GraphQL resolver. The server receives id of the user who uploads an image and a file which is the image being uploaded. The *file* holds its name and data stream. The stream is read and then

```
@Mutation(returns => UploadedFile)
async uploadProfileImage(@Args() {file, userId}: UploadUserProfileImageArgs): Promise<UploadedFile> {
  const { filename, stream } = await file;
  const fileRead = stream.read();
  const uploadedFile = await writeFile(`./uploads/${filename}`, filename, fileRead);
  await this.userRepository.update(userId, {image: uploadedFile.filePath});

  return uploadedFile;
}
```

Fig. 44 Profile image upload - code

the file is stored on the server. It is followed by updating a user database record with a new file path.

8.4 User Story 18

In case that the application becomes very popular and many users add statuses, it might be hard to find relevant statuses in such a wide range of topics. Therefore, some filters have to be applied. At this point, statuses can be limited to a certain area defined by a radius around a user's location. The next significant aspect related to search is a status category.

The Product Owner and I came up with three basic categories – Coffee and Tea, Party and Sport. The categories are just rows in the Category table, so any meaningful category can be effortlessly added in the future.

A user can add a status with a category when adding a new status as available since the User Story 6 has been finished (Fig. 46). When the modal opens, all categories are fetched, and they are listed in a select dropdown. The following procedure is same as it was before, only a categoryId is added to the request.

Then statuses fetched in the feed and user profile contain also their category. Obviously, the category can be also edited as in the [User Story 9](#).

The categories can be now used to search for more concrete statuses (Fig. 47). They can be combined and empty as well. If they are empty, a WHERE SQL clause ignores categories and returns all statuses filtered only by the radius.

Fig. 46 Add status - category

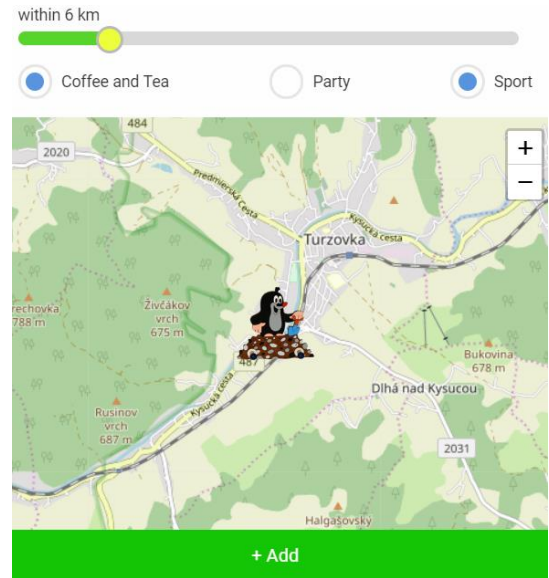


Fig. 47 Search by categories

8.3 User Story 17

Statuses in a user profile are very similar to those in the main feed (Fig. 45). The GraphQL and SQL queries have only a few differences. For user statuses, I need to get a `userId` from the current URL when the user is on the user-profile route and then use it to fetch only his statuses. The `React-router` package exposes all URL parameters, so the `userId` can be easily accessed. Another difference is that I do not need to filter the statuses by radius.

For its visual part on the front-end, I used the same React component for `status-item` as in the feed. When all statuses data is fetched, I provide that to the component also in the same way.

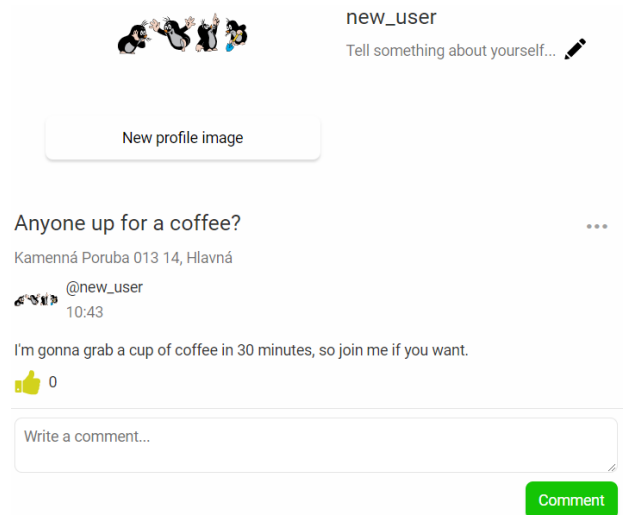


Fig. 45 User statuses

8.5 User Story 19

When adding a new status, a user has to either enter an address manually or choose to use a current location. It was good enough for testing and development purposes, but it would be quite tedious for users to fill out the entire form.

In order to make the whole process easier and faster, I used a map provided by Leaflet. Its API can detect various user interactions and gestures. It also allows to get latitude and longitude from map tiles that are already loaded. I combined these two Leaflet features, such that a user can select a status location by a right-click in the map (Fig 48).

When the map detects a right-click, a Redux action with coordinates is dispatched, the action then hits an epic that expects an action of that type (Fig 49). I used the same technique as for filtering by the radius, nothing happens unless a user does not click in the map for at least 700 milliseconds. That limits a number

of requests made for fetching an actual address with reversed geocoding. The same geocoding process is used when a user decides to use his current location.

Once the server sends a response containing an address, the Redux form is automatically filled out with data in the response.

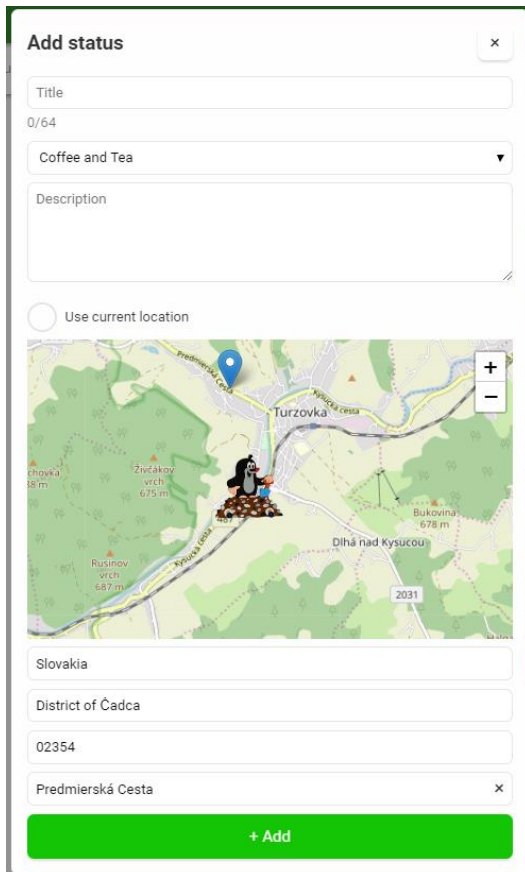


Fig. 48 Selection of location

8.6 Sprint Review Meeting

This was the last meeting before the final Sprint. First, we went through the last Increment alongside the user stories for the Sprint 4. The Product Owner was mainly glad to see the working functionality of status categories and getting the address from coordinates when creating a new status. He also liked to see a customized user profile.

For the lack of time, we agreed that the last Sprint should not involve anything rash. We both preferred to do a fewer planned user stories and rather check the functionality already delivered features. This means that we would not take any risk for having unstable and half-done features at the time of delivery.

8.7 Sprint Retrospective

Went well

- Leaflet package provides powerful and easily usable API, which is also well documented
- Generic components, such as form, form elements, modal dialog, saved some time

```
const getAddressFromMap: Epic<IReduxAction, any> = (action$) => action$.pipe(
  ofType(setNewStatusLocation.type),
  debounceTime(700),
  tap((setLocationAction) => {
    const {lat, lng} = setLocationAction.payload.location;

    fetchingAddress.dispatch(true);

    graphqlClient.query<{geocodeReverse: any}>({
      query: geocodeReverse,
      variables: {
        latitude: lat,
        longitude: lng,
      },
    }).then(response => {
      setAddress.dispatch(response.data.geocodeReverse.address);
    });
  }),
  ignoreElements(),
);
```

Fig. 49 Fetching address for coordinates

- High number of implemented features makes it easier to add new ones because they often share similarities

Could be improved

- It is quite tedious to update cached Apollo data on mutations. I believe there could a better way how to handle it.
- Geocoding service used for Molehill is sometimes not very precise. However, it is understandable for a free open-source service.

9. Sprint 5

9.1 Introduction

At the last Sprint Review Meeting I discussed plans for this Sprint with the Product Owner. The result is a list of user stories below. The User Story 20 is the only one kept from the original list of user stories created at the project establishment. The main reason for these changes is risk minimization before the project delivery. Because two out of three user stories are related to testing and improvement of existing code, the overall risk is reduced. Anyway, the most important core functionality of Molehill has already been implemented.

User stories – Sprint 5		Story points
20	As a member, I can mark a status with “join”, so that other users can see that I am coming to the status address.	8
21	Improve test coverage (technical).	12
22	Test performance of React components (technical).	3

9.2 User Story 20

Each status can have comments and likes. That should increase interaction and visibility among users. This user story’s aim is to further extend the same idea. If a user can see who is planning to attend a certain status, that fact should motivate others to join the status as well.

For this feature I had to create a new database table “Attendance” to store a new data entity. For now, I will keep it simple only with necessary columns – id, userId,

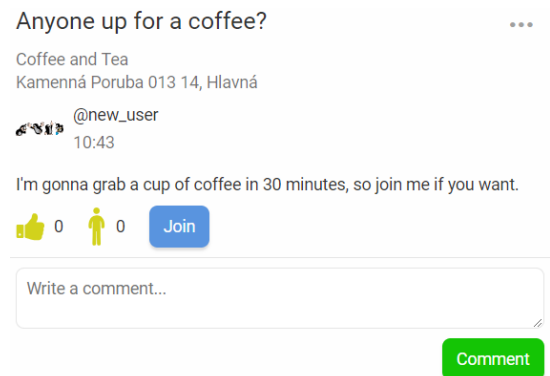


Fig. 50 Status attendance

statusId and createdAt. The number of entities has grown, so I decided to sketch a relational database model (Fig. 51).

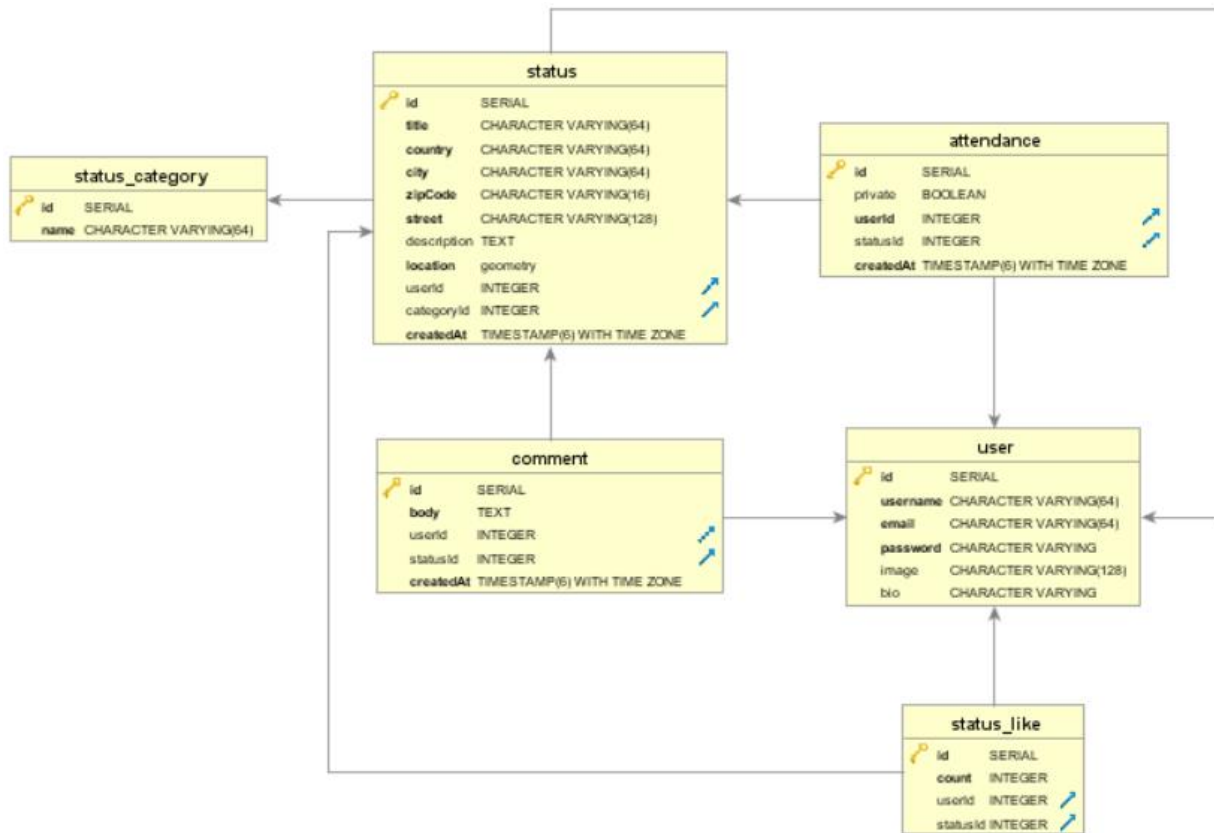


Fig. 51 Database model

A user can simply join any status by clicking on the “Join” button (Fig. 50). It opens a dialog window where the user has to confirm his decision to take part. After it is confirmed, a new Attendance record is created, and the Apollo cache is updated. The person icon next to the “Join” button shows the total number of people attending.

If a user has already joined the status, he can leave it with almost the same process. The Join button changes into “Leave” and that action also has to be confirmed.

When a user clicks on the person icon or the number next to it, a modal window opens (Fig. 52). The modal content has all users with confirmed attendance.

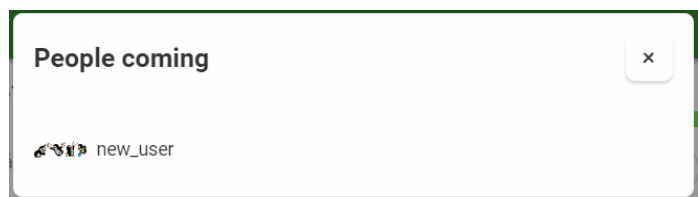


Fig. 52 Status attendance - overview

9.3 User Story 21

I wrote unit tests throughout the whole project period but now I will try to improve code test coverage. The most popular testing library for React applications is *Jest*. The performance and tools are most suitable because they both come from Facebook.

```

it('should render Button', () => {
  const wrapper = shallow(<Button />);
  expect(wrapper).toMatchSnapshot();
});
    
```

Fig. 53 Test - Button snapshot

The first specific kind of unit tests involves snapshots (Fig. 53). A snapshot (Fig. 54) is a text representation of a rendered component created when the test is executed for the first time. The test runner checks if the output is a correct JSX syntax and could be rendered. When the test runs next time, it checks if the output is very same as the one saved in the text document.

```
exports[`Button should render Button 1`] = `
<styled.button>
  <Styled(Styled(styled.div)) />
</styled.button>
`;
```

Fig. 54 Button snapshot

Another unit test specific for a React application with the styled-components package can be seen in Fig. 55. This test runs two assertions, first its snapshot is either generated or checked against existing one, and then its css property of *background-color* is checked against the value of background in Theme.

```
it('should render StyledContainer', () => {
  const wrapper = renderer.create(
    <StyledContainer theme={Theme} />
  ).toJSON();
  expect(wrapper).toMatchSnapshot();
  expect(wrapper).toHaveStyleRule(
    'background-color', Theme.background,
  );
});
```

Fig. 55 Test - StyledContainer

A more common type of unit tests can be seen in Fig. 56. It uses the typical AAA pattern (Arrange, Act, Assert). First, I define a constant variable *validEmail* (Arrange) with a string value that is in the format of valid email. Another constant variable *isValidEmail* holds a value returned by a function that is tested (Act). The last line in the function block checks if the variable *isValidEmail* is equal to *undefined*, which should be true (Assert).

```
it('should recognize a valid email address', () => {
  const validEmail = 'valid@email.com';
  const isValidEmail = email(validEmail);

  expect(isValidEmail).toBe(undefined);
});
```

Fig. 56 Test - valid email address

Jest comes also with a tool for code coverage reports. The entire project has 90 unit tests and 39 snapshots, which covers 83.69% of the code (Fig. 57). I tested mainly reusable generic components and functions that could cause errors throughout a large part of the application.

```
Tests:      90 passed, 90 total
Snapshots: 39 passed, 39 total
Time:       56.148s
Ran all test suites.
```

File	% Stmts
All files	83.69
__tests__	100
setup-tests.js	100
src	100
root-epic.ts	100
src/components	85.15
button.tsx	72.22
container.tsx	100
screen-center.tsx	100
spinner.tsx	100
svg.tsx	100
user-image.tsx	100
user-leaflet-icon.tsx	100

Fig. 57 Test coverage

9.4 User Story 22

React is a library for building components with fast performance. Its performance is very similar to other popular frontend frameworks, more specifically, Angular and Vue. However, it is a valid fact only if it is used with recommended practices from React's developers and its huge community experienced with large-scale applications.

My main concern was whether the React components are updated only if it is really necessary. I decided to use a Google Chrome extension called *React Developer Tools*¹³ for this purpose. The extension allows to debug React applications or just components right in Chrome DevTools (Fig. 58). It shows a hierarchy of all currently rendered components and all Props and Context available in a selected component. The values can be changed directly in the browser, if they are not read-only, which is reflected immediately in the application view.

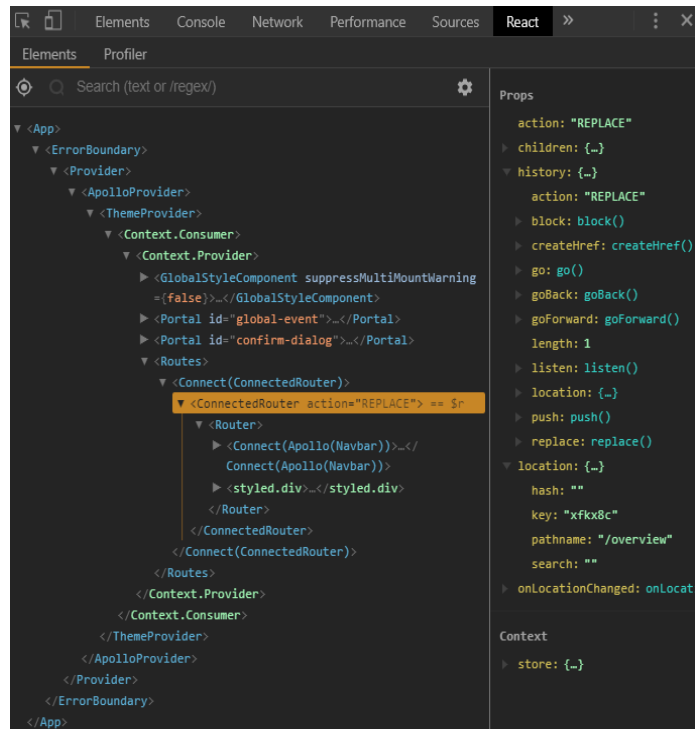


Fig. 58 React Developer Tools

The extension provides also other useful feature to monitor updates of components, which can be enabled by selecting "Highlight updates". When a React component is updated caused by changed props passed to it, the extension highlights the React component's border. Fig. 59 shows that the input field value was updated while I was typing the email address.



Fig. 59 React component updates

Unnecessary component updates can be eliminated with a help of the Reselect library. Its function *createSelector* takes a primitive values or other computed values processed with *createSelector* and caches its result (Fig. 60). It means that the final value is not recomputed if the provided arguments have not been changed. Its significance is seen mainly with more computed calculations, but in the context of React it does not update props which prevents a component from re-rendering a DOM node.

```
export const getRadiusInMeters = createSelector(
  getRadius,
  (radiusInKilometers) => radiusInKilometers * 1000,
);
```

Fig. 60 Selector

Another way how to reduce re-rendering is creation of class-based components by inheriting *React.PureComponent* instead of *React.Component* (Fig. 61). The advantage of *PureComponent* is in its automatically added

```
export class PrivateRoute extends React.PureComponent<Props> {
  constructor(props: Props) {
    super(props);
    this.renderRouteComponent = this.renderRouteComponent.bind(this);
  }
}
```

Fig. 61 React PureComponent

¹³ <https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en>

lifecycle hook that shallowly compares values of props. Hence, if the component receives new props, the hook compares the new props with previous ones, and if none of them have been changed, component update is prevented.

9.5 Sprint Review Meeting

At this meeting I could demonstrate only one new feature – attendance of statuses. The Product Owner liked to see that the application had a new way how users can see and know more about each other. The new feature made the system more interactive.

Then we looked into the unit tests with snapshots. The Product Owner could see how the tests were written and run. The code test coverage could be higher but at least the generic components were covered well. The rest should be improved in the future.

Optimization done in this Sprint was mainly recommended by the Product Owner who had experience with a large system, that was supposed to run not only on the latest high-end devices, but on devices with quite limited resources as well. I showed him what I had done to improve the performance and that was definitely enough for a project in this stage.

9.6 Sprint Retrospective

Went well

- Jest was already set up and ready to be used for the User Story 21
- I think it was a good decision not to rush through the last Sprint with too many risky user stories

Could be improved

- It was quite hard to write some tests that required mocks. I should spend more time investigating best practices for various kinds of tests.

10. Conclusion

10.1 Reflection

The development of Molehill application required me to apply what I have learned about software development during various courses over the period of four semesters. In addition to that, I also used acquired knowledge and skills from the mandatory internship.

My topic was focused on development of a software product, which involved mainly coding, but the chosen methodology was a great help. The most notable characteristics, I would point out, are effective time management and valuable collaboration with the Product Owner.

Some of the technologies used for this project were well known to me, so I did not have to start at ground zero. But my previous experience with back-end technologies, unit testing and everything related to the location API, geocoding and libraries for maps was quite poor. After reading documentations, articles, forums, I applied that straight away in Molehill, and now I feel more confident in this aspect for future projects.

After all, the delivered product meets almost all requirements formulated at the beginning of the project. The last Sprint was adjusted with new user stories and some of the planned user stories have not been finished, but I am honestly surprised that was the only change, when we consider that the agile development was used. Taken into account the final product and its development overall, this project was a valuable and successful experience.

10.2 Evaluation

The extremely frequent, technical feedback in regular intervals was undoubtedly helpful for quality of the delivered product. Estimated level of difficulty and time consumption of individual features in the form of story points helped me plan and organize work in advance. On the other hand, meetings with the Product Owner would have been more beneficial if he had had more time allocated for them.

The lack of time for meetings with the Product Owner had a few drawbacks. We could have done design sketches together to build potentially more user-friendly UI and UX. In my opinion, the Product Owner would be able to interfere with the development even better if there was more time to present top-level diagrams of the application features.

Technology-wise I am satisfied with the stack I selected. The chosen libraries/packages have extensive documentations and support from the community. Despite that, sometimes it was complicated to use certain libraries together because they were dependant on packages of different versions. Another difficulty was TypeScript types, which were in a few cases incorrect or missing. That means the initial choice of technologies is very important in a long run.

10.3 Next Steps

- The application needs only hosting to make it publicly available. From that point, it would be a minimal viable product that could be tested by first real users. It would be important to gather user feedback and make refinements to improve UX. Real users would be also a great source of suggestions for new features.

- Molehill was developed as a platform for connecting people in a certain area. The target group could be better specified, such that future development can focus more on attractive features for people with more specific characteristics.
- Application development and hosting cost certain amount of financial resources, therefore it is important to think about the best way how to monetize it.
- Now any user can create a status with a malicious intention. That should be eliminated with a verification system.
- Maps and services used in the application are provided by third-party developers for free under their conditions. They could be replaced by paid alternatives to get a better experience. Some providers offer maps that use SVG instead of map tiles. It results in smoother zooming and animations in general.
- If a user wants to get navigated to a selected status, it is done externally by Google Maps. However, the best option would be to keep the user in the application by a paid navigation service.
- After the in-app navigation from the previous point would be implemented, new features related to that could be added. For instance, users could get suggestions what statuses can be visited along the way to a selected status.
- Some more features to keep users active could be implemented. E.g., push notifications, recommendation system, user following.
- After the minimal version would prove to be successful among users, a mobile version could be built. It could be a hybrid mobile application simply built with Cordova or a native mobile application built by rewriting React visual components with React Native.

11. Bibliography / Webography

- <https://graphql.github.io/learn/>
- https://wiki.openstreetmap.org/wiki/Main_Page
- <https://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf>
- <https://www.agilealliance.org/glossary/xp>
- <https://www.typescriptlang.org/docs/home.html>
- <https://docs.npmjs.com/>
- <http://typeorm.io/#/>
- <https://19majkel94.github.io/type-graphql/>
- <https://www.apollographql.com/docs/>
- <https://www.postgresql.org/docs/>
- <https://postgis.net/documentation/>
- <https://expressjs.com/>
- <https://www.styled-components.com/docs>
- <https://jestjs.io/docs/en/getting-started>
- <https://airbnb.io/enzyme/>
- <https://react-leaflet.js.org/>
- <https://reactjs.org/docs/getting-started.html>
- <https://redux-observable.js.org/>
- <https://redux.js.org/>