Bedkom – An implementation of a reactive web application

INF219 spring 2019

Jonas Folkvord Triki

Abstract

In this report, we are going to tackle the obstacles of implementing an application consisting of a user-friendly web interface, as well as an API for business logic. We will address the problems with arranging company presentations, why it is a problem and our solution to it. In particular, we are going to take a look at technologies such as Haskell, Elm, React, TypeScript and Node.js/Express for automating the process of administrating and signing up to company presentations. To finish up, we will evaluate and discuss our results, and come up with a conclusion.

Contents

1   Introduction

This is a project for building a presentation management system for Institute of Informatics' student organization. The system consists of a back-end server with a NoSQL database and a front-end web application. The web application communicates with the API and takes care of authentication and business logic.

1.1   Motivation

The student association at the Department of Informatics, University of Bergen, echo, consists of many subgroups, with one of them being the business relations committee, or Bedkom for short. Bedkom has existed for several years and is mainly responsible for arranging company presentations for informatics students at the University of Bergen.

Company presentations are presentations where companies get the opportunity to tell students what they are working with. The companies view the presentations as a recruitment mechanism. The invited companies may also host workshops or give a lightning course on a current topic. After said presentation is done, the companies usually take the students out to an eatery (or orders catering services to where the presentation takes place) to get to know the students on a more personal level.

Company presentations are very much liked by the students but a hassle to manage for the committee members. It has become obvious that we need an automated system to cut down the manual work. The manual work required includes tasks such as creating multiple Google Forms, posting registration links on Facebook/echo.uib.no, ensuring that there are no unauthorized students, managing waitlist and finding replacement people to fill the spots after cancellations.

It is also a problem for the companies that contact the representatives in the business relations committee, because there is no centralized place for frequently updated information about the presentations and because communication is through emails, which makes the feedback loop rather slow.

We have also talked to a couple of students about the project. We got the feedback that well-organized and frequent company presentations give a competitive edge to the University of Bergen as a place of study in Norway.

Talking to the Bedkom committee members, they state that the process of organizing company presentations is old and needs to be improved.

1.2   Possible solution

An automated system (such as a website) could solve Bedkom's problems of poor presentation management. At a high level the system would offer the following functionality, organized according to the different kinds of users to the system.

1.2.1 Students

- Ability for students to register/deregister for company presentations at ease.
- Letting the students sign in using Mitt UiB / StudentWeb / Feide credentials, requiring little to no effort from the students.

1.2.2 Companies

- Ability for companies to get a full overview (audience, location, etc.) over the presentation they are going to give.
- Ability for companies to request presentations at given semesters.

1.2.3 Bedkom committee members

- A more structured workplace for Bedkom committee members with admin pages.
- Pages for creating, editing and managing
    - Presentations
    - Companies
    - Menus
    - Users
    - News

1.2.4 General features

- Shared login for the students, the companies that give presentations and the Bedkom committee members
- Easy shortcuts on the home page to see the next presentation or the latest news
- Presentations page with next presentation, upcoming presentations and previous presentations
- An about page for reading more about members, getting to see who sits in the committee and a contact form for the general public

Finally, we aim to develop a system that has a modern, sleek looking, user-friendly graphical interface.

As a summary, the goal of the project is to create a custom content management system for managing Bedkom presentations. The system consists of the client web application that connects to an API, which handles all the business logic, such as communicating with the database.

1.3   Goals for the project

At the start of the project we set some initial goals we wanted to achieve, both for the web application and the back-end server. The goals for the server implementation were as follows:

- Testable code.
- Secure, using modern techniques to solve authentication.
- Fast and scalable, since we would like the server's initialization time to be as close to zero as possible.
- Easily maintainable and well documented structure.

The server should implement an API that clients can use to communicate with the server.

With the API in mind, we wanted the web application to act as a view that just calls on the API with network requests. We came up with the following goals for the web application:

- Easy to use for the end-user, especially during registration for company presentations
- Easy for a Bedkom committee member to administrate presentations
- Available in multiple languages; internationalized
- Testable
- Well documented
- Securing input fields and user session data

We set a few concrete deliverables for the project. As a minimum requirement, the following features should be a part of the application before the end of the project:

- Easy login for both students and companies
- Registration form for company presentations for students
- Basic overview of presentations for companies
- Admin page for committee members being able to administrate content

In addition, the application should be secure and not exhibit unnecessary delays or lagging.

2   Requirements Analysis

This section describes the detailed requirements for the Bedkom system.

We did not have a particularly structed way of approaching the requirements gathering, but rather relied on past experiences with arranging and administrating company presentations with companies over the past semesters. Concretely, the requirements elicitation was done through multiple internal meetings with Bedkom. The stakeholders for the project are the committee members of Bedkom, students represented by echo and companies. Both students and Bedkom members were represented in the elicitation meetings, but we did not involve companies' representatives.

We identified the functional requirements for the Bedkom system.

2.1   Functional requirements

- Logging in
  - The system should have a combined place where student, companies and Bedkom committee members can log in.
  - The system should let students and Bedkom committee members sign in using their Mitt UiB/StudentWeb/Feide credentials.
  - The system should follow echo's guidelines and only allow students from the University of Bergen that are taking either
    - a bachelor/masters-degree in informatics
    - a bachelor's degree in ICT
    - a bachelor's degree in cognitive sciences with specialization in informatics
  - The system should automatically fetch information about a student upon first login and ask for the students e-mail address and if they have any allergies.
  - The system should provide companies login information upon creating a company presentation with them.
  - The system should offer a "Forgot my password" page for all users.
  - The system should offer the option to sign out at any time.
  - The system should verify the students every semester to ensure they are eligible to attend the company presentations.
- Users
  - The system should uphold the uniqueness of each user.
  - There should be different roles:
    - Student – an ordinary student
    - Company – contact person/employee at company

- ■ Bedkom – Bedkom committee members
- ■ Bedkom Admin – Bedkom head/deputy head
- ■ Super Admin – Head of echo
  - o The system should differentiate between the different user roles. Students and companies have their own set of permissions, but do not share details with each other. Bedkom has all the permissions student and company has, Bedkom Admin has all the permissions Bedkom has and Super Admin has all the permissions Bedkom Admin has.
  - o The system should offer the users with different roles, depending on their permissions in the system.
- • Loading screen
  - o The system should fetch the user's session, as well as the user's information while it shows the loading screen.
  - o The system should fetch a list of the presentations, news and Bedkom committee members while it shows the loading screen.
- • Pages
  - o The system should provide Bedkom committee-only pages, other users cannot access them.
  - o Home page
    - ■ The system should let the user register for the next company presentation.
    - ■ The system should show the user the latest headline of Bedkom news and offer the user to expand the headlines to articles.
  - o Presentations page
    - ■ The system should let the student register for any company presentation.
    - ■ The system should show the company presentations the student has registered for, as well as future and past presentations.
  - o For companies' page
    - ■ The system should provide companies a contact form for inquiries.
  - o About Bedkom page
    - ■ The system should show the committee members of Bedkom and provide a contact form for Bedkom related inquiries.
  - o Admin page
    - ■ The system should restrict the "Admin" page to Bedkom committee members and users with higher permissions.
    - ■ The system should provide sub-pages for each and every committee member related functionality:
      - • Presentations
      - • Companies

- Menus
- User
- News
  - The system should provide the same basic functionality for each sub-page:
    - Creating a new entry
    - Editing an already existing entry
    - Deleting an entry
- Presentations
  - After a presentation has been created, the system should automatically schedule posts on echo's webpage/media platforms to advertise the presentation.
  - The system should send a confirmation mail to a student after company presentation registration/deregistration.

2.2 Non-functional requirements

In general, we aim at readable, concise and well-documented code. For this project, we place special emphasis on avoiding code duplication, refactoring code when necessary, good/descriptive variable and method names and easily extendible code.

2.2.1 Back-end

- Login secured using a strong key derivation function
- Only authenticated access to data only
- Uploaded files should be stored securely
- API available through HTTP endpoints

2.2.2 User interface

- Readable and comfortable text (contrast between background and text)
- Neutral and appropriately sized font
- Simple page navigation with menu bars
- Animations during downtime, such as waiting for loading
- Easy to navigate to profile and change language
- Fast loading speeds
- Input fields are sanitized

2.2.3 Documentation

- API documentation for the client
- Should give insight into why the code is written the way it is

2.3 User stories

This section spells out the most important user stories for the system. User stories are an informal, natural language description of features, written from the perspective of the user. They follow a particular sentence structure (Agile Alliance, 2019).

2.3.1 For students

- As a student I want the web application to have me sign in using my credentials.
- As a student I want the web application to show the next company presentation such that I may register for it.
- As a student I want to see the latest Bedkom news such that I am updated if there were anything to happen to a company presentation.
- As a student I want the web application to be easily available through the internet.
- As a student I want to have a nice overview over the presentations I have attended.
- As a student I want to be able to send inquiries to Bedkom if necessary.

2.3.2 For companies

- As a company I want to be able to log into the system to get an overview over the coming company presentations I am going to hold.
- As a company I want to send inquiries to Bedkom regarding company presentations.

2.3.3 For Bedkom committee members

- As a member of Bedkom I want to be able to organize company presentations from start to finish.
- As a member of Bedkom I want to be able to upload the contract between me and the contact person the company to the presentation in the web application.
- As a member of Bedkom I want to be able to see who is registered for a company presentation.

- As a member of Bedkom I want to be able to deregister people from a company presentation.

2.4   Use cases

A use case diagram gives a high-level view of how different users might interact with our system.
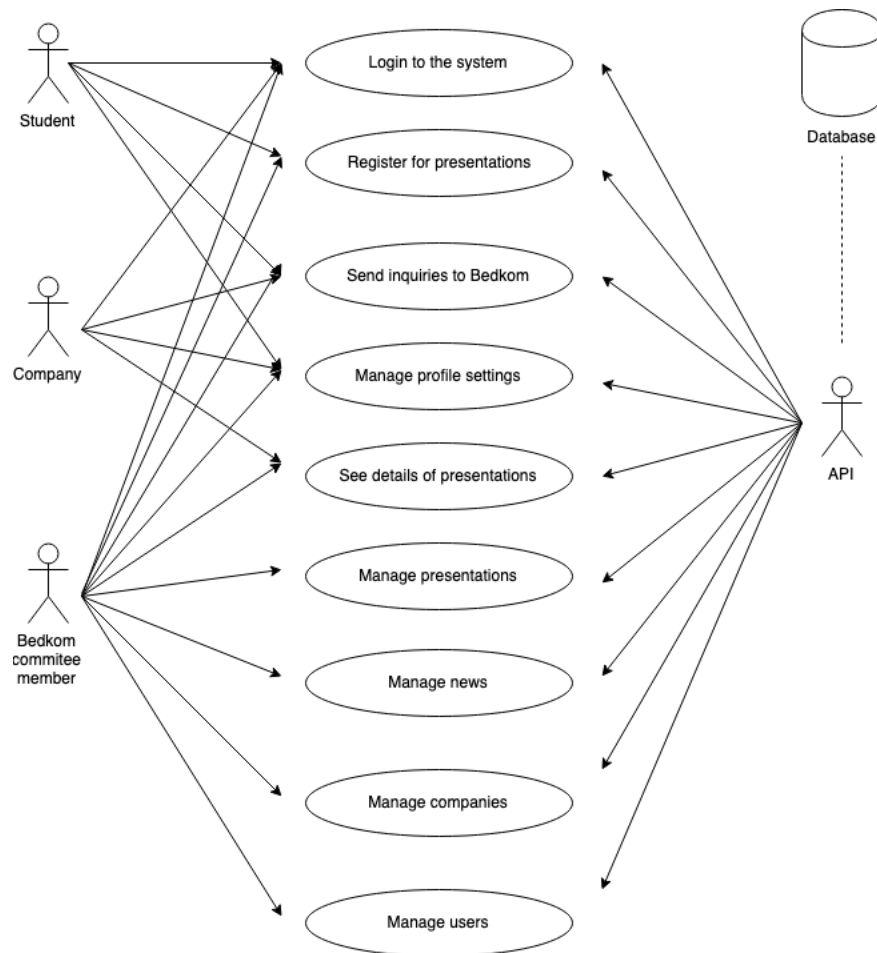

Figure 1: Use case diagram of system

3   Design

This section describes the design of the Bedkom application to satisfy the specified requirements. The visual design was performed using mockups and logical design through modelling, using tools to help in the process.

3.1   Front-end

3.1.1 GUI

For the GUI design, we first outlined the different screens that can appear in the application and how they are related to each other. We then created a mockup for most of the screens using Adobe XD. Figure 2 shows the different artboards of the GUI.

Figure 2: Adobe XD artboards of Bedkom GUI

We put some effort into making the application aesthetically pleasing giving the application a professional look and feel. We chose two primary colors throughout the application, dark blue (primary color) and yellow (accent color). We aimed for pages with a sleek look and as few distractions as possible.

To give a glimpse of the visual design, we inspect a few of the oft-encountered screens. The login screen is shown in figure 3. This is where the student starts, by signing in the application using their username and password.
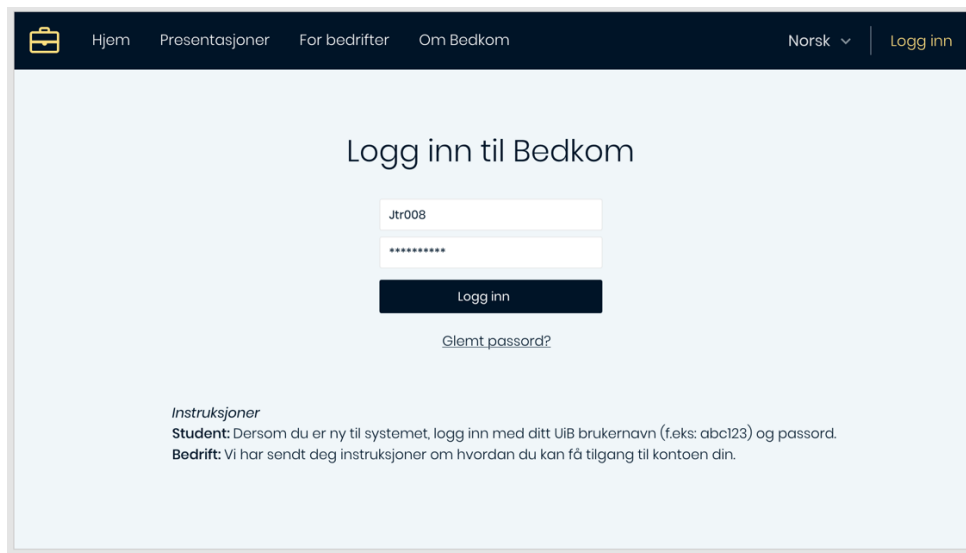
Figure 3: Bedkom login-screen

After the student has been logged in, it is be possible to register to company presentations. The registration screen is shown in figure 4.

Figure 4: Bedkom presentations page

After the student has registered to a company presentation, an e-mail will be sent to confirm the registration. It is also possible for the student to go back to the presentation page to deregister before a set deadline.

3.1.2 Architectural design

We loosely follow the Model-View-ViewModel architectural pattern (Vincijanovic, 2018), or MVVM for short, for design of the front-end code. The reason for selecting

MVVM is that it is considered a good approach to achieve modularity in GUI implementation. It also fits well with the React ecosystem that we use in our implementation.

MVVM has for main blocks (as seen in figure 5):

- View – UI layer that the user interacts with
- ViewController – has access to ViewModel and handles user input
- ViewModel – has access to the Model and handles business logic (read: API-calls)
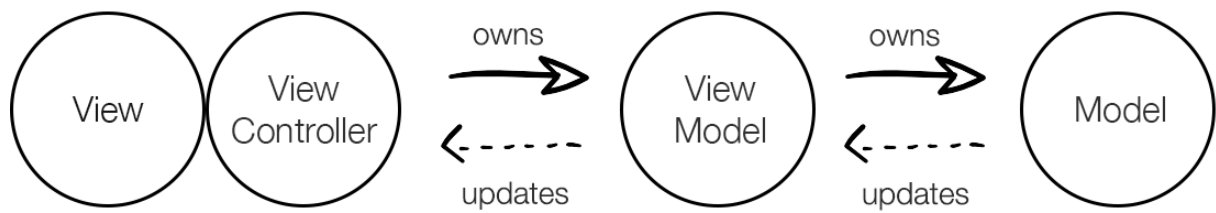- Model – application data source

Figure 5: MVVM diagram (Vincijanovic, 2018)

3.2   Back-end

To serve as a foundation for the applications database schema, we modelled the back-end data as an ER-diagram, as shown in figure 6.
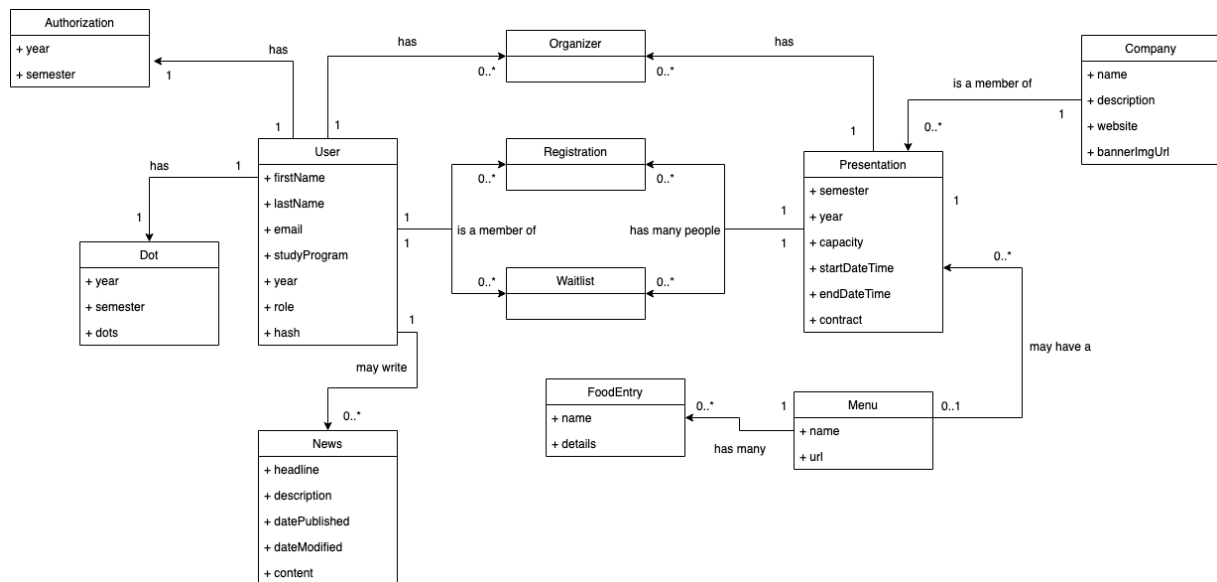

Figure 6: Database relations

Following is a brief explanation to each entry in the diagram:

- **User**: The student/committee member
- **Authorization**: What year/semester the student was checked for eligibility to use the system
- **Dot**: Punishment-system for students, i.e. not showing up to presentations/late deregistering
- **News**: Bedkom articles published by committee members
- **Presentation**: Company presentation
- **Organizer**: Organizer (Bedkom committee member) of the presentation
- **Registration**: A student registration to a particular presentation
- **Waitlist**: After the registrations reaches its limit, the waitlist takes over the registration
- **Company**: Company that presents itself during a presentation
- **Menu**: Food menu to let participants choose what they want to eat after the presentation, if the company so wishes to take the participants to eat somewhere
- **FoodEntry**: Food entry in the menu

3.3   Technology stack

A modern web application must rely on a host of technologies. The Bedkom system uses TypeScript as programming language, Node and express for back-end server API, React for front-end web application and AWS for storing persistent data. We describe the relations of the technology stack, as illustrated in figure 7.
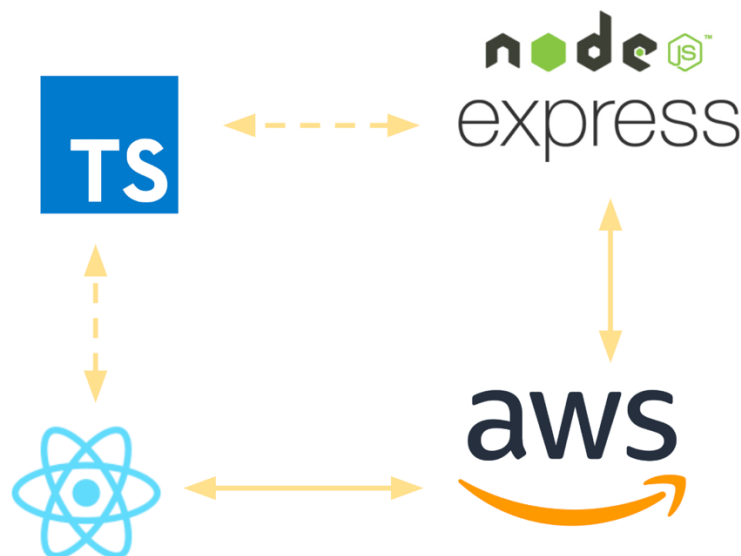

Figure 7: Technology stack

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. We used TypeScript with Node.js, an asynchronous event driven JavaScript runtime. Node is designed for building scalable network applications (Node.js Foundation, 2019). We decided to stick to Express for handling HTTP requests and routing in of endpoints in general. Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications (Node.js Foundation, 2019). As for persistent storage, we chose Amazon DynamoDB, which is fast a flexible NoSQL database service.

For the back-end, we used Node.js and express with TypeScript. We implemented interfaces in TypeScript for each database entry in our ER-diagram; the structure of the database is visible in the structure of the code.

For the front-end, we used React with TypeScript. React is a JavaScript library for building web pages and acts as an extension of the language, bringing in concepts such as JSX (JavaScript XML). JSX is an extension for XML-like code for elements and components, which makes it easier to create components throughout the whole application.

3.4   Deployment

We used Amazon Web Services (AWS for short) for deploying the web application/API to the internet and persistent storage. AWS is a subsidiary of Amazon that provides on-demand cloud computing platforms on a metered pay-as-you-go basis (Amazon Web Services, 2019). In particular, we used the following services:

- Amazon DynamoDb – persistent storage, NoSQL database
- Amazon S3 – for saving files and serving the web application
- Amazon Lambda – for serving the back-end API

During the implementation, we worked on our local machines and did then not have to deploy the application to the internet. We discuss deployment in the Discussions and Conclusions section.

3.5   Exploration of alternative implementation technologies

The initial plan of the project was to create a fully functional system using only functional programming languages. We considered Haskell for the back-end API and Elm for the front-end. The reason we selected Haskell and Elm to be the initial languages, was the desire to apply functional languages in practice.

We started out with creating a REST API using Servant in Haskell. Servant is a set of packages for declaring web APIs at the type-level and then using those API

specifications to write servers, all in a type-safe manner (Sevant, 2019). To connect to the database, we used esqueleto, a type-safe domain specific language for SQL queries. Esqueleto introduces a new syntax of writing queries which closely resembles SQL.

Example esqueleto query which filters by PersonName

```
select $
  from $ \p -> do
    where_ (p ^. PersonName ==. val "John")
return p
```

Which in return generates the following SQL

```
SELECT *
FROM Person
WHERE Person.name = "John"
```

As one of the tasks, we implemented script to verify if the student is represented by echo and eligible to attend company presentations. To check for if a student was eligible to attend company presentations, we created a custom script which, given Feide username and password, signs the student into StudentWeb, scraps the webpage for studies and returns back with the result. To do this in Haskell, we used a package called "tagsoup", which parses and extracts information from HTML documents, in addition to "wreq" for doing network HTTP requests.

We did not progress further than that with Haskell. It became evident that the time it took to create, test, and debug functions in Haskell was not worth the effort. Types in the different networking libraries are very complex, and combining them is often not trivial. Often error messages from the type checker are very long and confusing leading to development feeling like a fight with the compiler.

The experience with Elm was similar. We did not come much further than a "Hello World" application in it.

4   Implementation

We implemented the system using two separate code bases, one for the front-end and one for the back-end, to clearly separate the modules of the system. We utilized Git's version control system to keep track of files and go back to old implementations if needed.

4.1   Back-end

We use npm to manage the back-end code base, and follow a typical structure of a Node.js backend with TypeScript.

4.1.1 Scripts

The npm run script are customizable scripts to automate repetitive tasks, for example for building our project. We added the following scripts:

```
"scripts": {
  "prebuild": "tslint -c tslint.json -p tsconfig.json --fix",
  "build": "tsc",
  "start": "NODE_ENV=dev ts-node-dev --respawn --transpileOnly
./src/main.ts",
  "deploy": "serverless deploy --stage dev",
  "deploy:prod": "serverless deploy --stage prod",
  "test": "echo \"Error: no test specified\" && exit 1"
}
```

The "build" script is self-explanatory and first runs the TypeScript linter, tslint, and reports back if there are any errors.

The "start" script allows for faster development and simply transpiles/interpreters our TypeScript code, using the ts-node-dev package. It also restarts the node-process after we have made changes to our code, which came in handy.

The "deploy" script uses Serverless (Serverless, Inc., 2019) to deploy our code to AWS Lambda. We did not use this script a whole lot, since we worked locally anyway.

To run the scripts, we simply ran "npm run <name>" in the terminal and replaced <name> with the name of the script. The "pre…" scripts are being ran before the main script. It is also possible to define "post" scripts.

4.1.2 Dependencies

We are using the following npm dependencies:

```
"dependencies": {
  "argon2": "^0.20.1",
  "aws-sdk": "^2.418.0",
  "cheerio": "^1.0.0-rc.2",
  "connect-dynamodb": "^2.0.3",
  "cookie-parser": "^1.4.4",
  "cors": "^2.8.5",
  "csurf": "^1.9.0",
  "dynamoose": "^1.6.4",
  "express": "^4.16.4",
  "express-session": "^1.16.1",
  "express-validator": "^5.3.1",
  "jsonschema": "^1.2.4",
  "morgan": "^1.9.1",
  "multer": "^1.4.1",
  "request-promise-native": "^1.0.7",
  "serverless-http": "^1.9.1",
  "uuid": "^3.3.2",
  "winston": "^3.2.1"
}
```

Following is a brief explanation for each dependency:

- argon2
  - Used for hashing/verifying hashed passwords. Argon2 is a key derivation function and was the selected winner of the Password Hashing Competition in July 2015 (Khovratovich, 2019).
- aws-sdk and dynamoose
  - Used for communicating with our DynamoDb database.
- cheerio and request-promise-native
  - Cheerio was used to scrap web pages a crucial package for reverse engineering the Feide-login. Request-promise-native was needed to keep track of the cookies/session data as we reverse engineered the login.
- cookie-parser, cors, csurf, morgan and winston
  - Middleware functions used for parsing cookies, handling cross-origin resource sharing, setting up CSRF-protection on critical endpoints, logging endpoints and logging in code respectively.
- express
  - Framework for handling HTTP requests
- express-session and connect-dynamodb
  - Middleware for enabling user sessions and saving the session to a table in DynamoDb.
- express-validator and jsonschema

o   Middleware functions used to validate input from user, as well as a function for validating json object structures.

- multer
    o   Middleware function used to capture files for uploading with multipart/form-data.
- serverless-http
    o   Wraps around our express app and allows us to deploy the API to AWS Lambda through Serverless at ease.
- uuid
    o   Helper function for generating unique ID's for entries in the database. We are mostly using uuidv4.

As for the directory structure of the code, we decided to split each endpoint into its own file and endpoint sub-paths into their own directories. An endpoint is the URL where our back-end server can be accessed by the Bedkom web application. In figure 8 we get a glimpse of how the back-end server's endpoints are structured:
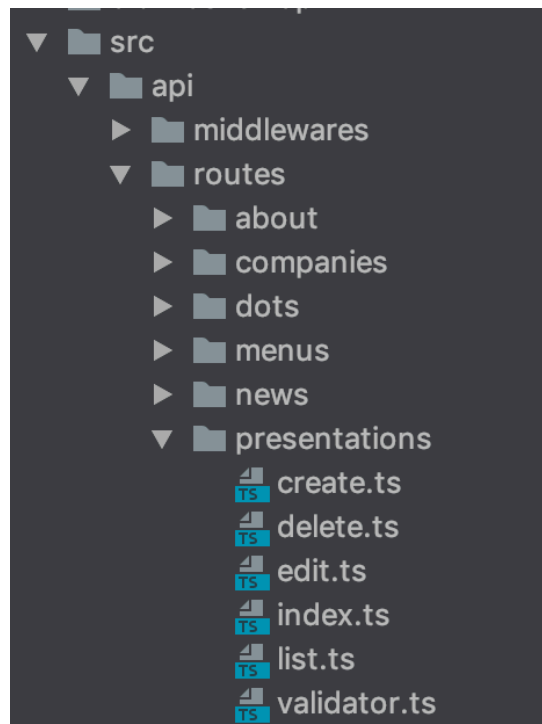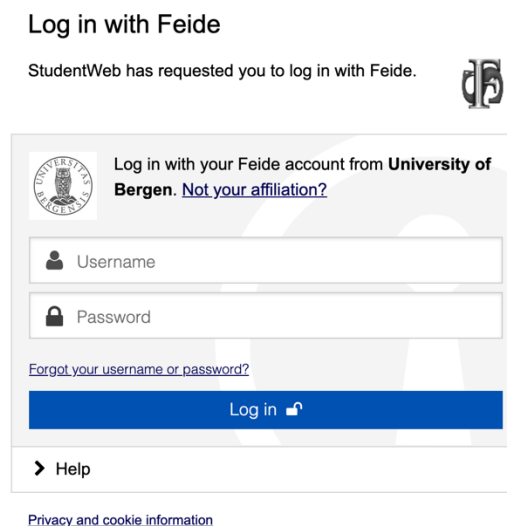
Figure 8: API's directory structure

4.1.3 Authentication mechanism

The authentication mechanism is important part for us during the implementation. We use argon2 to salt and hash passwords before the hashed string is being stored in the database.

21

With the requirements in mind, it was important to fulfill echo's rules of which students are allowed to attend company presentations. We did not find an external API we could use for verifying the eligibility, so we made our own login-script using reverse-engineering and web scrapping.
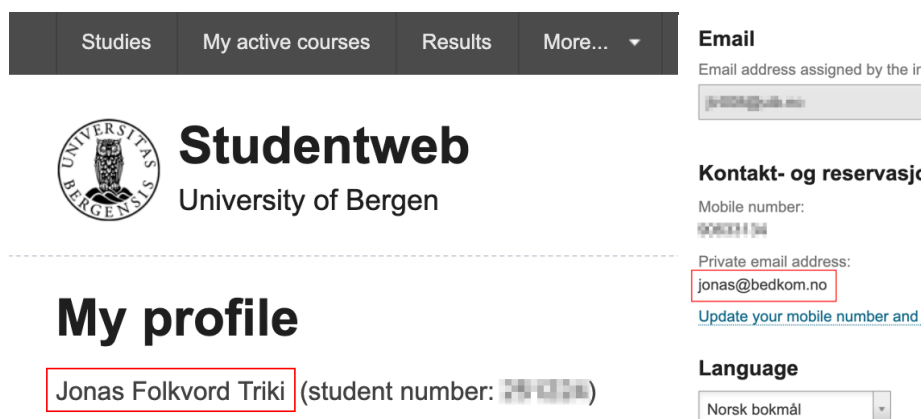
Our login-script first logs the student into StudentWeb with Feide, as shown in figure 9:

Figure 9: Feide login

After the student has been authenticated through Feide, our login-script navigates to "My profile" and scraps the page for the student's name and e-mail, as shown in figure 10:

Figure 10: My profile scrapped fields (red rectangles)

Our login-script then it navigates to "Studies" and scraps the page for the current study of the student, as shown in figure 11:
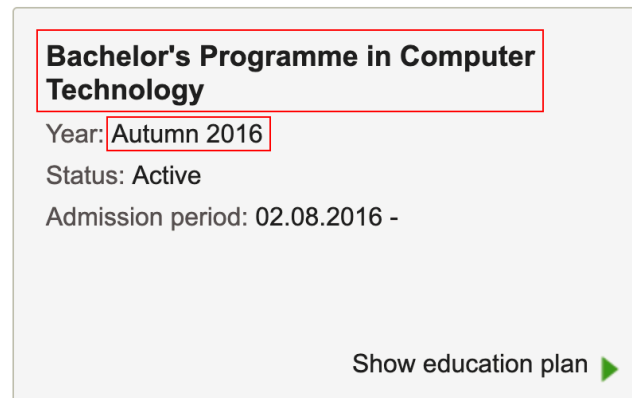
Study programmes

Figure 11: Study programs from StudentWeb

Once the script has fetched the name, e-mail and current studies of the student, our login-script responds back if the student is eligible to attend company presentations.

4.2   Front-end

The font-end follows a typical structure of a React application with TypeScript and similar to the back-end, we use npm to manage the code base.

4.2.1   React state management

React has components which let you split the UI up into independent pieces, and think about each piece in isolation (Facebook Open Source, 2019). Components in React are conceptually just JavaScript functions and may have "props" (which stands for properties) which is just arbitrary inputs. Similar to props, components in React may have "state", but it is private and fully controlled by the component. Following is an example of a React component with the current date as state:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
```

```
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
    </div>
  );
  }
}
```

To get the current state of the component we have to access the this.state(…) object and to set the current state of the component we have to call this.setState({…}). Now, imagine a component with a dozen of states to keep track of; it becomes messy after a while. The introduction of React Hooks thankfully solves this problem for us.

React Hooks (Facebook Open Source, 2019) as introduced quite recently. React Hooks let you split one component into smaller functions based on what pieces are related. Here is an example from the Login page in our web application, using React Hooks:

```
const [username, setUsername] = useState('');
const [password, setPassword] = useState('');
```

What happens here is that we first call the "useState" React method in the code, which creates a hook with the initial value "" (empty string). TypeScript also notices that we initiated with a string and now knows that "username" and "password" should have the type string as well.

We are now able to call setUsername/setPassword in the code, updating the internal state of the Login component.

4.2.2 Internationalization

The web application is indented to be used by both Norwegians and by others who do not speak Norwegian. It is then only natural that one should be able to switch language while using the web application. To solve this problem, we used a module called "react-intl", which is an internationalization module created by Yahoo. We chose this module because it is being currently maintained and is among one of the most popular modules for internationalization. What is nice about this module, is that all translations directly are defined in the program code, either through JSX or separate TypeScript files.

We also used a particular babel plugin, "babel-plugin-react-intl", for extracting string messages for translation from modules that use React Intl (FormatJS, 2019). In conjunction with this babel plugin, we also used "react-intl-translations-manager", a npm module for managing translations that the babel react-intl plugin extracts for us. The pipeline is as follows:

1. Babel React Intl plugin extracts translations from code and saves them in a temporarily folder.
2. React Intl Translations Manager then picks up the extracted translations and creates JSON files for all the languages we would like to support.
3. Temporarily folder created by babel plugin gets deleted.
4. React starts compiling TypeScript and runs the web app.

As an end result, the user can switch between languages should they so desire. Since we wrap our main component, the "App" in a "IntlProvider" component, switching languages do not lead to a refresh of the webpage. The IntlProvider passes down translations to all child components, its child components and so forth.

4.2.3 Sharing state between components

We recall from the react state management section that React has components which lets you split the UI into isolated parts. This unfortunately comes with a disadvantage when we would like to share state between components, due to how the React application is organized as shown in figure 12:
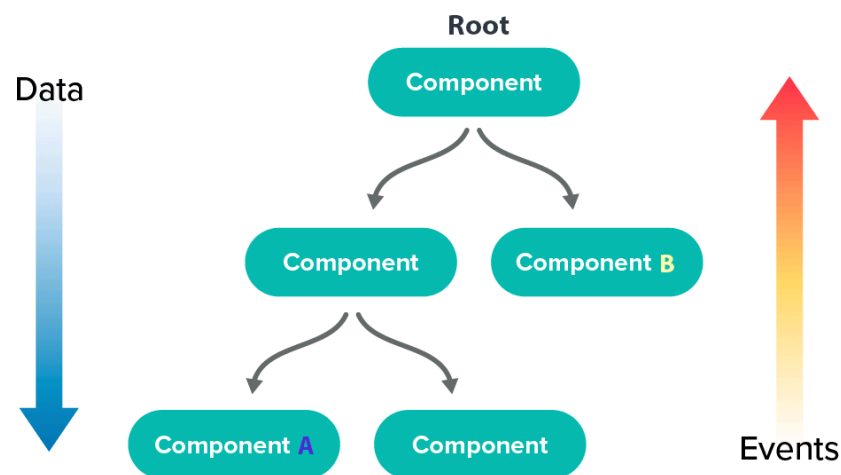

Figure 12: React component tree

To give an example of the problem: component A has loaded some critical data and needs to share it with component B. What component A first has to do is call some event all the way from component to component up to the root. The root then might send down the data as props to component B. This involves a lot of unnecessary steps and creates unnecessary boilerplate code for the programmer. To solve this problem, we are store the global state of the web application in one place, using React Redux (Abramov, 2019).

Redux is a state container for JavaScript apps and a pattern for managing the applications state. It is very beneficial in our application, since we have many

components managing their own state, and sharing the state from left to right (as shown in figure 12) becomes very hard to manage at scale.

What the Redux pattern does for us is to have one shared store where the global applications state is stored, and is accessible to every child component of the app. The components simply dispatch actions, which in return do something to the state (in the reducer), update it and re-render components that need re-rendering. This is shown in figure 13.
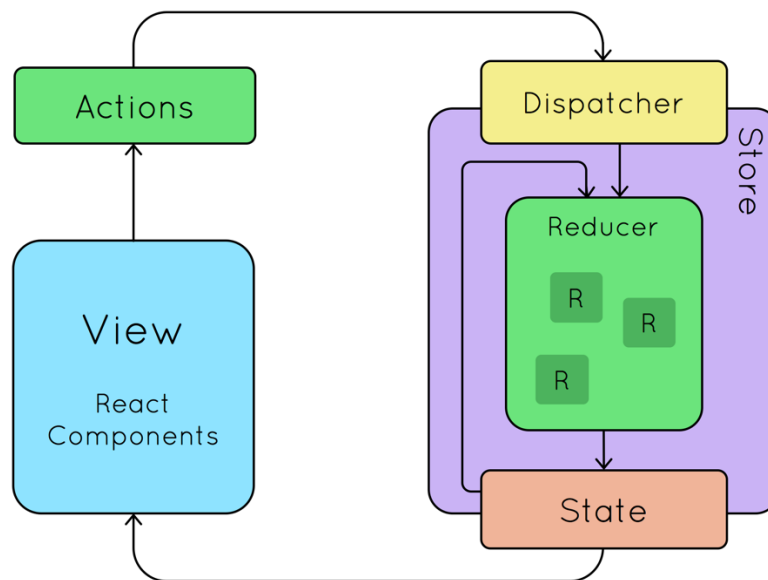

Figure 13: React Redux state-flow (ESRI, 2019)

We also used Redux Saga (Abramov, 2019) as a middleware framework to Redux for catching actions before they are sent to the reducer. What is nice about Redux Saga compared to other middleware frameworks such as Redux Thunk (Redux JS, 2019), is that we do not end up in "callback-hell"; we can test our asynchronous flows easily and our actions stay pure. Redux Saga also uses one of ECMAScript's (JavaScript language specification) newest addition, the Generators.

4.2.4 Styling components

For styling the HTML elements in our web application we used Styled Components. Styled Components let us create styles very explicitly without defining separate .css files. They also ensure that we have no class name bugs and they are painless to maintain.

4.3 Status

The web application as is, is currently not ready for public use. It still lacks core admin functionality, such as creating/managing presentations and users. We have come a long way and the application will be ready to be used in a few weeks. The core back-end API endpoints are implemented and are ready to be tested.

We have completed most of the functional requirements as specified in the Requirements Analysis. We chose only to implement some pages and certain functionalities, due to time limitations.

Following is a list of what has been implemented/is yet to be implemented.

Functionality	Status
Login	Implemented - Login for students and Bedkom committee members. Not yet implemented - "Forgot my password" functionality
Users	Implemented - Separation between the users' roles
Loading screen	Implemented - Loading user's session/information while it shows the loading screen. - Fetching of public data (presentations, news and Bedkom committee members) while it shows the loading screen.
Home page	Implemented - Showing the latest headline of Bedkom news Partially implemented - Registration for the next company presentation Not yet implemented - Expanding the headlines to articles
Presentations page	Not yet implemented - Registration for any company presentation - Showing the company presentations the student has registered for, as well as future and past presentations.

For companies' page	Partially implemented - Contact form for inquiries
About Bedkom page	Implemented - Showing the committee members of Bedkom Partially implemented - Contact form for Bedkom related inquires

Admin page	Implemented
	
	- Restricted to Bedkom committee members and users with higher permissions
	
	Partially implemented
	
	- Companies page, creation of new company entry
	
	Not yet implemented
	
	- Presentations page
	- Menus page
	- Users page
	- News page
	- The same basic functionality for each page:
	    o Creating a new entry
	    o Editing an already existing entry
	    o Deleting an entry

5   Discussions and Conclusions

We have now discussed our problem briefly, that company presentations are a hassle to manage manually, and that the process of creating and administrating presentations may as well be automated. We also discussed our implementation of a web application connected to a custom-made API and evaluated our work. Further, we discussed what we learned and things we could have done better for another time.

We have worked with a humongous number of technologies, spanning from very low-level web scraping libraries such as Cheerio to high-level styling libraries such as Styled Components. In particular it was interesting to implement our own session management and endpoint protection on the server in the back-end.

If we were to do this project at another time, we would certainly spend a little more time on revisiting the workload and maybe reducing the size of the project. The workload of this project exceeded our expectations, due to a high, unexpected time usage on features and components. We would also spend more time on documenting the code, both in the front-end and the back-end. Currently, the back-end API is undocumented, which is seen as a bad practice. For the architecture of the React application, an architectural model we did not get time to look at was MVVM, which was mentioned in the design section.

We would like to end on the future work required to complete the project. What is great about how we designed the Bedkom system, is that the code is modular. If one would like to implement a new endpoint in the back-end API or create an entirely new page in the web application, it should be straight forward as to how one attacks the problem. Looking at other endpoints and pages, the developer will get a sense of how to take this project further. For more details as to what to implement further, one would have to re-visit the Status section under Implementation.

At some point, we have to deploy the web application, which we did not get to since we worked mostly locally. We would also include the following AWS technologies:

- Amazon CloudFront – content delivery network for the web application/API
- Amazon API Gateway – for managing the API
- Amazon Route 53 – for managing the DNS and domain names

To help us deploying the API to Amazon Lambda, we used Serverless. Serverless is a framework which abstracts away all the AWS configuration for deploying Lambda functions.

The source code for the project is available through two GitHub repositories:

- **React Application**: https://github.com/JonasTriki/bedkom
- **Express API**: https://github.com/JonasTriki/bedkom-api

We would also like to thank Jaakko Järvi (Järvi, 2019) for supervising this project.

6   Sources

Abramov, D. e. a., 2019. *Redux.* [Online]
Available at: https://redux.js.org/
[Accessed 27 May 2019].

Abramov, D. e. a., 2019. *Redux Saga.* [Online]
Available at: https://redux-saga.js.org/
[Accessed 27 May 2019].

Agile Alliance, 2019. *Role-feature-reason.* [Online]
Available at: https://www.agilealliance.org/glossary/role-feature/
[Accessed 27 May 2019].

Amazon Web Services, 2019. *AWS.* [Online]
Available at: https://aws.amazon.com/
[Accessed 27 May 2019].

ESRI, 2019. *React Redux Overview.* [Online]
Available at: https://www.esri.com/arcgis-blog/wp-content/uploads/2017/09/react-redux-overview.png
[Accessed 27 May 2019].

Facebook Open Source, 2019. *Components and Props.* [Online]
Available at: https://reactjs.org/docs/components-and-props.html
[Accessed 27 May 2019].

Facebook Open Source, 2019. *Introducing Hooks.* [Online]
Available at: https://reactjs.org/docs/hooks-intro.html
[Accessed 27 May 2019].

FormatJS, 2019. *Babel plugin React Intl.* [Online]
Available at: https://github.com/formatjs/babel-plugin-react-intl
[Accessed 27 May 2019].

Järvi, J., 2019. *Personal webpage.* [Online]
Available at: https://www.ii.uib.no/~jjarvi/
[Accessed 27 May 2019].

Khovratovich, D. a. A. J.-P., 2019. *Password Hashing.* [Online]
Available at: https://password-hashing.net/
[Accessed 27 May 2019].

Node.js Foundation, 2019. *Express.* [Online]
Available at: https://expressjs.com/
[Accessed 27 May 2019].

Node.js Foundation, 2019. *Node.js.* [Online]
Available at: https://nodejs.org/en/about/
[Accessed 27 May 2019].

Redux JS, 2019. *Github.* [Online]
Available at: https://github.com/reduxjs/redux-thunk
[Accessed 27 May 2019].

Serverless, Inc., 2019. *Serverless.* [Online]
Available at: https://serverless.com/
[Accessed 27 May 2019].

Sevant, 2019. *Sevant homepage.* [Online]
Available at: https://www.servant.dev/
[Accessed 27 May 2019].

Vincijanovic, D., 2018. *MVVM architecture.* [Online]
Available at: https://medium.cobeisfresh.com/level-up-your-react-architecture-with-mvvm-a471979e3f21
[Accessed 27 May 2019].