6 Implementation

This section covers the implementation of the application Join Denmark. The chapter will firstly cover the technologies we have used and why we have chosen them, secondly the use of components and pages are covered and thirdly the personalisations that can be done by the user. While this chapter covers most of the implementation the chapter 8.1 Join Denmark covers the adaptations to the implementation after testing the initial prototype. Hence the prototype might contain more than what is described in this chapter.

6.1 React

The application is implemented using React. Choosing to use React was primarily in order to gain knowledge and experience with it. It is mostly recommended for large web applications. React is a JavaScript library and is used for making the view part of the application - that is the user interface. The smart thing about React is that the page is not reloading but the content renders. This makes it faster and scalable which might be a problem to achieve in larger applications. Another reason is that if we were to continue working on this project after finishing the bachelor it could become much bigger than what it is today. React further allows us to reuse components which we have used a lot and will be expanded upon in the chapter 6.5.2 Components. Some of the benefits of reusing components are saving time as we have to write less components and it makes maintenance significantly more simple, the app gets a more consistent behaviour and look and each component has it's own logic. React also supports browsers on mobile phones which was another pro as we wanted primarily to develop the application for phone use.

One of Reacts core concepts is state. State is essentially representing changeable parts of each component in an application. Comparing to what we know from Object Oriented programming it is similar to an object that has properties that tells us about said object.

6.2 Redux

During the code development we learned that it can be quite tricky to manage the state in react when wanting to use it globally. We either had to keep passing it between components using state which quickly gets complex or we had to find another solution. The reason for this was that we had to implement a login feature and quickly foresaw trouble with user related data and managing the state of the user being logged in. We found Redux and after reading about it figured that it was a good solution to the problem due to the simplicity compared to React useContext. Additionally, it would save us a lot of trouble passing state back and forth between parent and child components. Redux ensures that state is always up to date and therefore correct.

Redux is a React library which provides an architecture to manage data flow in the application i.e. used for state management in applications. Setting up Redux requires creating a store, the source of the current state. In order to change the state an action must be dispatched. The action is a JavaScript object with the change. Tying the state and action together is done with a reducer which is a function that returns a new state or "the next state" of the app. The reducers are also called pure functions and is a rule for using Redux. The pure functions return the same output every time they are given the same input. The state is immutable only and there is no other way to change state than dispatching an action. By having state managed like this it becomes much easier to maintain. Redux is quite complex and came with a steep learning curve, however, managed to improve our application significantly and saving us a lot of time due to the simplicity.

6.2.1 Persist-Gate

Even though Redux solved many of our challenges with global states, one remained. Whenever a user would refresh the app the state would be reset to initial state and the user would not be logged in. We used Redux Persist, more specifically the component Persist-Gate. With the implementation of Redux Persist the Redux object is saved in a persistent storage which means the user is not logged out on refresh, further the user will stay logged in when closing the browser too. The user will need to actively log out otherwise the state is persisted in the storage. This is a benefit as the user will not have the hassle of logging in every time they have to use the app, but they will already be logged in and the personalised view will be displayed.

6.3 Database

The application needed a database for storage of content, settings, users and so on. The Application we were building did not have a backend as such so we needed to either build

a backend or find a database service that could work as a backend (backend-as-a-service). We had looked at Amazon Relational Database Service (RDS) with AWS and Microsoft's Azure offers. As this is a school project and the application itself is not very big data-wise we decided that we could argue for using a free service. Luckily there are many services today that are offered for free or are free for students. We ended up choosing Firebase Firestore from Google. Firebase itself is a platform that has a number of products. One of them is Firestore which is a cloud hosted database. It is also realtime meaning the database synchronises any update right away, which turned out to be a big plus for us when testing. At last it is also offline first meaning that while it is supposed to be connected the majority of the time the app is in use it can cache data and sync once the user is back online. When doing research we found that there are quite an extensive amount of documention from Firebase which was a big advantage. Firestore is described as being targeted towards mobile and web applications which was a pro in relation to our application.

Further the database is noSQl which gave us an opportunity to try to work with this structure, which proved to be very flexible. As mentioned we have no previous experience working with noSQL but still found the flexibility of the tree structure to be an advantage. The databases is build of collections which has documents which then has fields. It is further possible to have collections for each document as well. We chose to not go this route partly because it was not necessary due to the minimal data we needed to write to the database and partly because a quick search showed that many users advised against it.

With the flexibility there are pros and cons. It is smart that we can create fields and collections in the code. A document or a document field in the database can be created with a value set in the code even if this field or document did not originally exist in the database. However, this also meant that testing provided some extra work as we could accidentally create fields or collections that we did not intend to.

While it has been rather simple to set up and figure out how to query the database there has been other limitations than the cons of flexibility. One limitation is the lack of ability to retrieve data from different collections that has a relation. This can be solved only by accessing all collections of the related documents.

6.4 Cache - memoize

After having set up the database and started using it successfully, we found that in many instances we were accessing the database several times reading the same information i.e. when re-rendering the pages. The prototype does not has a lot of data and the limit of reads and writes per day is not reachable given the app size so it might not seem like a big problem. However, iterating the same data over and over did not seem scalable and hence we decided to look for a solution. We talked about caching but as we have no previous experience we went to search for some help. We discovered that JavaScript has something called Memoization which solves this. It caches the returned values of a function and if we call the same function with the same value again it retrieves the values in the cache rather than executing the function again. Besides providing scalability for the future it also meant that if the user looses connection the loaded data can still be displayed when jumping around the application.

6.5 Page and Components

6.5.1 Pages

The implementation of the application consist of six different pages. The HomePage, the JoinDenmarkPage, a SubCategoryPage, a CategoryPage, a LoginPage and a SettingsPage. The HomePage is designed like shown in figure 6b in chapter 4.6 Changes/adapting feedback. The HomePage is the first page the user sees when the user opens the application. The JoinDenmarkPage will be elaborated upon in chapter 6.8.3 Join Denmark. The SubcategoryPage is a page with tiles similar to the HomePage. This is intended for categories that have similarities. They can then be gathered in a subcategory page. An example is the public sector category. Figure 3b and figure 3c in chapter 4.1.3 Category pages illustrates the design of the category page which is the page where the content related to the chosen category is presented to the user. Chapter 6.6 Implementation of the CategoryPage will go into details about the implementation of this page. The last two pages are the LoginPage and the SettingsPage. On the LoginPage the user can choose to login with either google or email. Chapter 6.8.1 User login goes into details about how the login works. The user gets access to the SettingsPage after logging in to the application. On this page the users can change their settings and logout. Chapter 6.8.2 Settings will go into details with the SettingsPage.

6.5.2 Components

Each page is built of seperate components. A component is a piece of React code which returns a React element. This concept is known from object orientated programming where elements are called objects. Components are reusable which means that one component can be used one or more times on one or more pages. An example of a component is the Tile. The Tile is used several times on the HomePage and used several times on a SubcategoryPage. A Tile is given a headline and a picture and returns a styled Tile. Other examples of components are the navigation bar (NavBar), the search bar and the HomeButton, which is reused on the majority of the application pages.

The use of components reduces the amount of code needed to be written since the code does not need to be dubpicated for each component. Additionally, components increases the scalability of the application because they are reusable and can be generated for each element in a collection. Some of the benefits of reusing components are that the application gets a more consistent behaviour, since each component has its own logic and the maintenance gets significantly more simple, since the code is not dublicated.

The tiles example above can be used to explain the scalability. The database (see chapter 6.3 Database) consist of several collections. One of the collections stores the needed information (headline and picture) of each Tile on the HomePage. When opening the HomePage the first time, the tile collection is loaded from the database, whereupon the tile information is extracted from each Tile in the collection and used to generate the Tiles. This means, that new Tiles can be added simply by adding the Tile information in the database.

6.6 Implementation of the CategoryPage

This chapter will go into detail with the article page and explain all the components associated with this page in order to give a detailed explanation of the implementation of a part of the application. The CategoryPage shows the articles related to a category. The CategoryPage is a good example of how simple we can build a page by reusing components. The page itself is even reused for each category in the database. The CategoryPage represents the content that renders when the user clicks a Tile either on the HomePage or SubCategoryPage.

6.6.1 Navbar, picture and homebutton

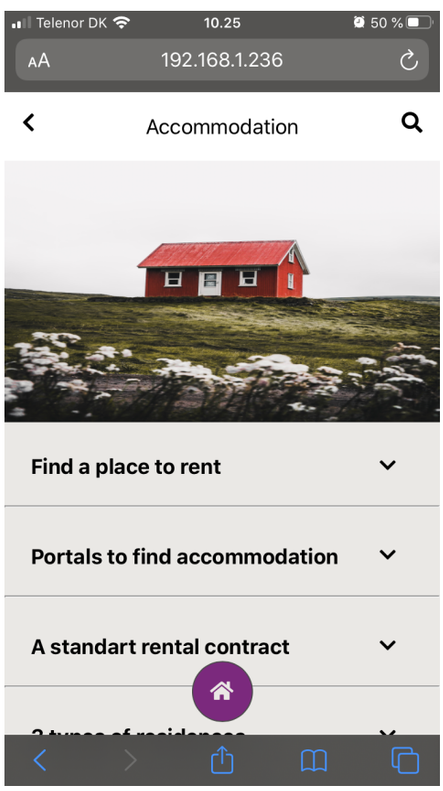


Figure 7: Accommodation page

The NavBar (short for navigation bar) is the white bar in the top of the screen. The NavBar is reused on all pages except the HomePage. On the left is a left arrow (indicating "back"), in the middle the title of the page on which the user currently is and to the right a magnifying glass (indicating search). The arrow indicating that the user can go back to the previous page is made by using the hook useHistory(). With this hook we can access a history object. The history object contains amoung other things the history of routes and a function called goBack() which simply goes back to the previous route.

The title in the middle is passed from the CategoryPage via state. The CategoryPage itself gets it passed via state from Tile. The reason for having a title is to make it easy for the user to identify where in the app they are.

The search function displayed with a magnifying glass can be clicked. Once clicked, the user is presented with an input field in which they can search among all content in the app. This way the user is not required to be in the exact right category in order to find the content they are looking for. Once the user types in words the results appear below as DropTiles, however, when the user presses the tile they are

directed to the page where the article belongs. I.e. if they are currently on the accommodation page and search for CPR they will be rerouted to the CPR page. For more details about search see chapter 6.7 Search.

The CategoryPage has a picture related to the content of the category. The picture is the same as on the tile the user has selected, in order to navigate to the CategoryPage, but in a non shaded version. This ensures some coherence for the user. Besides the coherence, the reasoning for including a picture on the CategoryPage is that it brings some life to a page and something that is visually easier to process compared to only having DropTiles with text shown. The link to the picture source is passed through state to the CategoryPage from the Tile.

The HomeButton is another component that is reused a lot throughout the application. As with the navbar, any page that is not the HomePage displays this button. As the house icon indicates the function is to links the user back to the home page from anywhere in the app. That way the user can always easily get back. The go back on the NavBar only goes one step back so if the user is further in the app using the home button can save clicks. Further the home button stands out to the user as it is purple whereas most of the page is beige.

6.6.2 Droptile

The DropTile is the tile displaying the articles and a little arrow down (see fig 7). It is a separate component. Each DropTile is generated by checking that there are any articles in the database for the given category and then setting the initial state of the droptile with the headline, author, subheading and body. The articles are loaded from the database with the hook useEffect() which sets the initial state of the articles. In DropTile we use a function to get a snapshot of the database and caching it using memoize. The cached articles will then still be available to the user if the user looses connection as they are now cached. DropTile is thus reused for each article easy due to the component already being build. It simply needs to be used with the required variables.

The DropTile has a text in the form of a question or headline which is passed via state from the CategoryPage. This text is the headline of the article from the database. It further has an arrow down which upon clicked opens the DropTile which then displays the author and the subheading of the article. Further, the arrow has changed to an arrow up indicating that the user can close the DropTile again. It is easy to get an overview of the articles when only the headline is displayed.

The author and subheading was passed to the DropTile via state from the CategoryPage. After the subheading text there is some blue text reading ”read more” which upon clicked expands the article showing the body. The body is given from state too, but it is formatted in DropTile. In the database the text is simply a long string and we needed to display the text in a more visually friendly way. Therefore, we chose to add a function which formats the text by splitting it on every ’#’ sign and returning each substring in a seperate ¡p¿ tag. The authors of the articles can add the ’#’ sign where they want a new line. If it was not formatted it would be hard to read, as the text would not have any line breaks.

The choice to have only a subheading shown upon clicking the tile was also a preference expressed among the users. By only showing the subheading the user can quickly assess if this was what they wanted to read. Otherwise, there is still many other headlines visible on the screen as opposed to opening the full body of the article which can be a lot to take in. As one can see in Figure 3 on page 6 there are several ways to implement the way to display the articles to the user. Another design that one user preferred was that the click on a DropTile took them to another page with all content displayed. The user would then have to click the back button in the NavBar in order to come back to the overview of the articles. The problem with this is that it requires more effort for the user when they have to determine if the content is relevant. There is much more visually to process when complete changing what is rendered. With the design we implemented it is simplified as the user can still see other articles and scroll when they have opened one.

6.7 Search

The search function is a valuable feature of the application. With the search function the user can easily find a specific topic. The search function consist of two parts. The first part is an algorithm which generates a searchable collection in the database. This chapter calls this algorithm the Search Generator. The second part is a front end component which returns the search bar and handle the functionality of this search bar. Our search function is inspired of Ken Tan’s article [3].

6.7.1 The Search Generator

In our research, we found that Firebase has an option for checking whether an array contains a specific value. We chose to build the search function upon this feature so we needed an algorithm to generate substrings of all headlines in our application and save the substrings in an array.

The code of the Search Generator is to be found in appendix D. We have added the code to the appendix since the code not is a part of the application itself. As mentioned in chapter 6.1 React, React is used to build the frontend part of an application. Therefore, our application does not have a backend part which is where the code of the Search Generator would otherwise have been be placed. Even though there is no backend, we chose to create the Search Generator as an executable file because it was the most effective way to generate the keywords when new headlines were added. The Search Generator is run by changing the security rules of the database so that it is allowed to write, whereafter you type in "node SearchData.js" from the Terminal while being in the correct folder. `addKeywordsToSearchDB()` on line 44-77 does.

The Search Generator is to be called after adding new headlines and articles to the database.

First this function deletes all the old content of the search database in order to clean it. This might not be the most beautiful way to do it, but it ensures that the search collection is up to date after the generator has run.

Next it goes through all the categories defined in the collection and gets all headlines in each collection from the database. Each of these headlines goes trough the `getAllSubstrings(string)` method which returns all the substrings of the headline as an array. To give an example `getAllSubstrings("hey")` returns h, he, hey, e, ey and y in an array. All the arrays with the substrings are saved together with the related headline and category in the search database.

6.7.2 The search component

The search component handles the search. When the user clicks on the magnifying glass icon in the upper right corner of the application an input field opens. When the user type something in this input field a search is performed. The search algorithm finds all the cases where the text typed by the user is equal to a string in the generated substring arrays in the search database. If there is one ore more matches the related headline is displayed as the search result in the user interface. If the user clicks on one of the search results the related category is used to redirect the user to the related page.

6.7.3 Scalability

As mentioned in chapter 6.3 Database, Firebase does not have a method to scan through all tables in the database. Therefore, we had to hardcode some of the search function in order to make it work. The result is, that the search function is not very scalable compared to the rest of the components in the application. There is two hard coded problems.

Firstly, if you add a new category in the database, you have to manually add the title of category to the array of collections in the Search Generator. After this, you have to edit the security rules of the database so you have access to write and manually run the Search Generator in order for the search generator to update the searchable collection in the database.

The second issue is found in the search component. A document in the searchable collection in the database consists of the headline, associated keywords and the title of the category. The document is therefore missing the path to the picture source and the link path which redirects the user. Consequently, we need to generate these. This is done by using the title of the category. The picture source is created by joining a string which consist of a link to the local collection of images, the name of the collection and lastly the string ".jpg". The result is that all pictures has to be named exactly the same as the article they belong to and the file format has to be jpg. In our application the image of the CPR category is of file format jpeg hence the search result does not have a picture. The link path which redirects the user is also created using hardcoded code. Actually, we have a helper function to generate paths, which we also use, but since some of the articles are placed in a subcategory we have to add the title of the subcategory before passing it on to the helper function.

We might be able to solve these problems by adding functionality to the Search Generator or by rebuilding our database so that the collections were optimised for searchable collections. But due to a lack of time and the risk of ruining some of the other functionality in the application, we chose not to implement this. Further we did not have a big focus on

building a backend for the application which could have helped greatly with this scalability issue.

6.8 Personalisation

As mentioned in chapter 5 The value of Join Denmark, one of the key values of the application is to make it personalised for the user. We have attempted to personalise the application by adding thea possibility to login, the possibility to hide categories with no relevance for the user and by having a JoinDenmarkPage progress page.

6.8.1 User login

The personalising of the application is dependent on the user choosing to login. We have chosen not demand that the user must login. This is because we want the content of the application to be available even though the user does not want to login. The login enables the user to hide irrelevant categories or show relevant ones. Further, the users have a possibility to track their process in the Join Denmark category if they are signed in.

The user login is implemented using Firebase authentication. We chose to use the Firebase authentication since we were already using Firestore from Firebase for our database and it was an additional service already offered. We found that sticking to using only one service was a good solution so we did not have to cross reference the authentication storage from another service with Firebase.

The first time a user logs into the application a user id is created both if they are using Google or creating an account with an email and a password. The user id is saved in a collection of users in the database. The only things saved in the database besides the user id are the settings the user has disabled and the process of joining Denmark which both initially are empty. We chose to save the id rather than i.e. the email address of the user due to privacy considerations. The user document in the database is therefore completely anonymous.

Every time a user logs into the application the Firebase authentication returns the user id. The id is used to load the user data from the database which get saved locally in the Redux store. This ensures that the user gets their personalised view. The user can sign out from the SettingsPage at which the user data will be removed locally from the Redux store.

We have added security rules to our database which ensures that unauthenticated users can not add content to the database. Additionally, the user id is used as a key which ensures that users can only change their own user document in the database.

6.8.2 Settings

Users have access to the SettingsPage after they have logged into the application. The SettingsPage contains several toggle buttons which give a possibility to disable categories. The disable categories are added to the document of the user in the database collection of users in the documents field called setting. At the same time the the Tiles on the HomePage and the SubCategoryPages are filtered so that the disable categories are not shown.

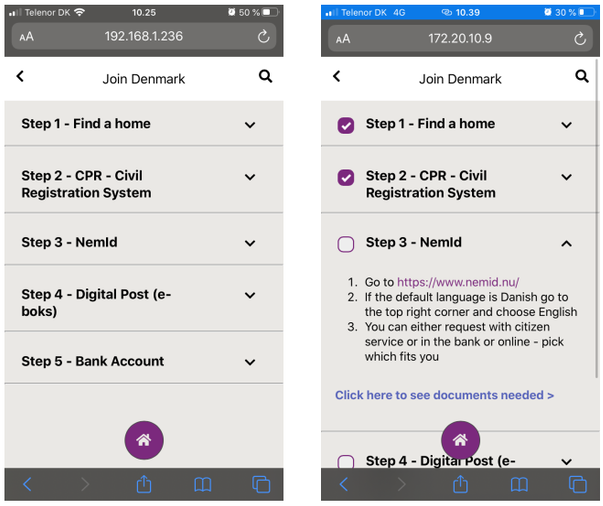
An example is that users can choose whether they are studying or not. If they choose to disable the Student setting then the State Education Fund (SU) category will be hidden. Users can always enable a category again and even though a category is hidden, the user can still search for topics within it. This is because it is only the Tile from the HomePage or a SubCategoryPage which gets hidden and thereby eliminates visual noise.

The possibility to hide categories does not make much sense in our application right now as it only have two pages with four tiles on each. But this feature makes the application more scalable as the application could have i.e. 50 categories in the future. Then it makes sense a user that not is studying can hide i.e. 10 categories related to studying in Denmark by one click on the Student setting.

6.8.3 Join Denmark

In the preliminary user interviews, as mention in chapter 3.1.5 UI, we found that users would like an option to track their process of joining Denmark by having a checklist stating what steps they need to take and in which order. The Join Denmark page is the implementation of this feature.

Like the article page the Join Denmark page consist of tiles where the content drops down when clicking on it, but while the tiles on the articles page are unsorted the tiles on the join Denmark page are placed in the order you need to solve the steps. Each step tile has a title and a list of steps which guides the user through the step. Lastly, a title has a list of documents needed in order for the user to solve the step.



(a) The Join Denmark page when the user has not signed in.

(b) The Join Denmark page when the user has signed in.

Figure 8: The Join Denmark page in tested prototype

The user always has access to the Join Denmark page. But if the user chooses to login it enabled the feature of tracking progress through the steps by ticking off the steps once the user have completed them. Figure 8a shows the Join Denmark page when the user has not signed in while figure 8b shows the page as user that is logged in, enabling checking off the steps the user has completed.

In our test of the prototype, described in chapter 7, we found a couple of things which we would like to improve on the join Denmark page. These changes are described in chapter 8.1.

6.9 Styling

We have used CSS to style our application. We have chosen to style our web application as an mobile application. This have been our focus since the beginning of the project because the users will typically use our application on their phone in order to get information fast. Additionally, we prefer using time on implementing new functionality rather than expanding the styling to cover tablet and computer screen. The application is predominantly styled responsive which means that the size of the components follow the width of the screen. The benefit of using responsive design is that the application fills the screen, both on small phone screens and big phone screens. The disadvantage is that some components i.e. the HomeButton gets to big when the screen gets to wide. The text in the application is not styled responsive so we have chosen a font size which works on phone screens. The text is therefore very small compared to the components when the screen gets very wide i.e. on tablet or computer.

We have tested the styling with the following mobile devices: Galaxy S5, pixel 2, iPhone 6/7/8 og 6/7/8 plus og iPhone X. These test were made using the develop mode of the google chrome browser. We have only used IPhone 6 and Iphone 8 when testing our application on a physical phone. During development we mostly tested the application in the chrome browser using develop mode.