Maynooth University
National University of Ireland Maynooth

PlanEat

CS 385 MOBILE APP DEVELOPMENT

Team Iron

Kevin

Justyna

Maiwenn

Ripandeep

Chapter 1:
Why did we develop PlanEat?

When we first met as a group and started brainstorming ideas, there wasn't really a consensus on what we wanted to do. Various planners, video-games, brain exercises, etc were discussed. There were creative differences, but we all agreed on one fundamental principle: we needed to be working on an app that has real utility for users, and that people would genuinely want to use for an extended period of time. Scalability was also an important factor - we wanted to get a core idea up and running relatively quickly with the option of adding more features.

Ultimately, we decided to make an app that could tackle the issue of meal planning and allowed users to discover new recipes, create a list of their favourite recipes, create plans based on these favourites, and to view the ingredients and steps needed to prepare these dishes. This perfectly fits our goals.

A meal planner app naturally appeals to a wide spectrum of people and we could create something that could genuinely help people who, for example:

- are tired of eating the same meals and want to prepare something more exotic.
- have particularly poor planning skills (e.g they forgot to buy all of the necessary ingredients).
- want to improve their diet and avoid processed foods.
- who just want to impress a date with an uncharacteristically good meal!

It didn't take long before we came up with the name, PlanEat (FoodLife and EasyEating didn't quite make the cut, and Good Food sadly already exists).

Once we implemented the basic feature set, we could get more creative and ambitious if time allowed us (e.g. laying out the nutritional value of different meals).

What does PlanEat do?

PlanEat provides users access to a very complete recipe API and an easy way to search for new recipes. Upon starting the application, users are brought to a homepage where multiple pages are selectable: "Search for Recipes", "Shopping List", "Favourites", and "Random Recipe".

It was clear in early discussions that we should use an existing API to generate recipes, ingredients, etc rather than creating our own JSON array. However, we needed to develop a good understanding of the available APIs which themselves introduced a lot of complications which will be explained later in depth.
On the "Search for Recipes" page, a user can select a search filter from a drop-down menu, and input any recipe that comes to their mind into the search box and tap the "Search" button. A list of resulting recipes will be returned and any search (both the search box input text and the search results themselves) can be immediately cleared by simply tapping "Clear".

Just allowing users to see the names of different recipes wouldn't be particularly useful, so from here users can:

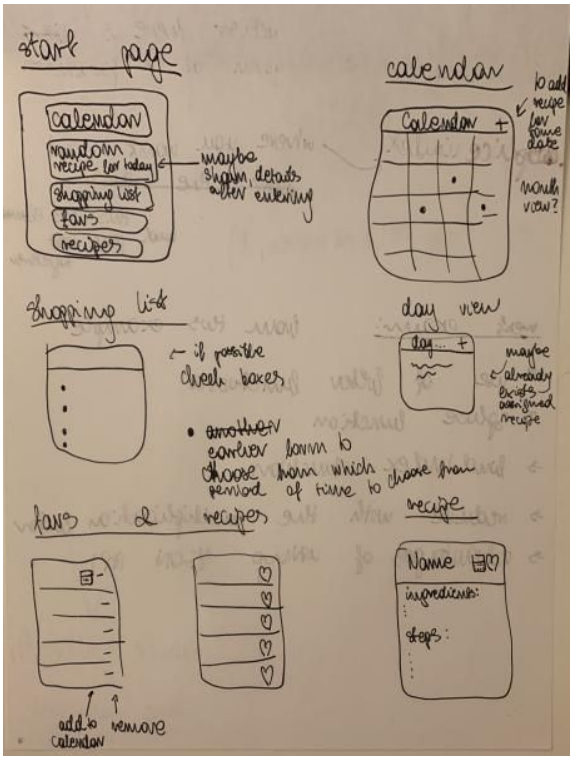
- add any recipe they see to their favourites by tapping the corresponding "Add to favourite" button
- check the details of the recipe (name, an appropriate image of the food, the amount of portions, preparation time, and the ingredients and steps required to prepare the food) by tapping the corresponding "Show Recipe" button.

On the favourites page, favourited recipes are listed and can be casually deleted by the user. Users can also load a random recipe which will be displayed in the same format as the "Show Recipe" page.

You can switch between these different components whenever you need to by using the navigation bar on the top of each page which is essentially all the buttons available on the homepage, and a "Home" button to return to this homepage.

Chapter 2:
How did we design PlanEat?

Let's now focus on the design of the application! When we initially discussed our project together, Justyna took the initiative and started work on some wireframe diagrams. We outlined the main functionalities that were essential to the project (search recipes, show recipes, switch pages, save to favourites, etc), and consideration was put in to make sure that these components would connect with each other in a logical way so that we wouldn't run into unexpected problems deep into the project.

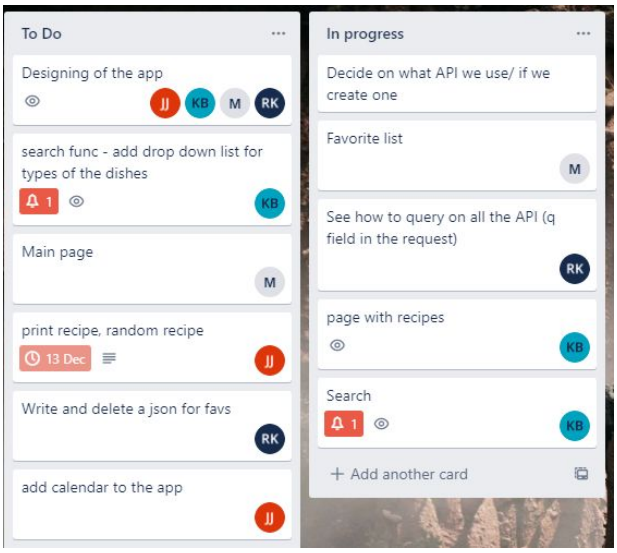


Once these diagrams were finished, we started assigning tasks to individuals on Trello. As a group, we researched how to make calls to the API (the syntax required). Once everyone was assigned a set of tasks, they suggested changes to the diagrams to reflect how they would personally like to implement their components.

The group

We decided that Maïwenn would work on linking the components together so that a user can transition between multiple pages through a navigation bar on the top of each page. She decided to first attempt this via conditional rendering and then use routing if there were any issues with this

approach (there were not). She additionally opted to take responsibility for the visual look of the app and to make it user-friendly as well.



Kevin took on the search component, intending to generate recipes from the input entered into a search box. Considerations had to be put into how the API query string would be updated due to limitations of the initially selected API. He decided that the proposed Edamam API offered insufficient API queries (5 calls per minute threshold) and another one (Spoonacular API which has a threshold per every 24 hours) was found although the limited API queries still caused headaches for the duration of the project. He also needed to include some filters, deciding that he would do this via a drop down menu and further dynamic adjustments to the API query string.

Ripandeep was assigned all storage responsibilities. He decided that it would be best to first attempt to store JSON files locally, and if issues arose, he would switch to using Firebase (this had to be done). Users expect modern apps to be synchronised and for small changes in their apps to be performed in the background so he felt that this approach was generally superior anyway. He also did the majority of the initial research into the API.

Justyna was initially tasked with implementing the calendar component. This task was almost complete but storage issues impeded our progress and the task was abandoned in favour of heavily adapting the existing search functionality to show the portions, an image of the food, the preparation time, and the ingredients and steps required for a particular recipe. Justyna also added a component to display a random recipe.

While the tasks were well divided out, they were not very strict as we regularly helped each other when bottlenecks were encountered. Once we all had some of our components done, Maïwenn linked all these separate pages together and worked on improving the visuals.

Our main goal, despite the complexity of certain aspects of the project, was simplicity for the user. We put in a lot of consideration to engineer this simplicity during the design process.

Chapter 3: How did we implement the functionality PlanEat?

SEARCH

To start with our API, one of the core functions that we wanted to get running early on was the search functionality using the Edamam and then the Spoonacular API. The Search component required a few different functions. The initial basic search was simple. We used **ComponentDidMount()** to fetch the json data from the API and store the results in an **apiData** variable which is then mapped to a table (rendered). Doing multiple custom searches was the tricky part.

The function used by the search box to update the API query string (**API_URL**) with the user's inputted search is **editSearchTerm(event)**. Here the inputted search is concatenated onto the necessary prefix ("**&query=**"), and this is concatenated onto the end of the default website string (**website** variable). An API query string is formed and just waiting to be used, e.g. "https://api.spoonacular.com/recipes/complexSearch?apiKey=4ef1662e8c6b4cd48502128b989b7000&query=mango"

```
//uses input in searchbox
//to alter API query
editSearchTerm = (e) => {
  const prefix = "&query=";
  const searched = e.target.value;
  const cuisine = this.state.cuisine;

  this.setState({ searchTerm: searched });
  this.setState({
    API_URL: this.state.website
      .concat(prefix)
      .concat(searched)
      .concat(cuisine)
  });
};

// sets "isTimeToUpdate" variable as being true
// so componentDidUpdate() will
// do another API call
performSearch = (e) => {
  this.setState({ isTimeToUpdate: true });
};
```

performSearch(event) is triggered when the "Search" button is pressed (onClick). This sets the **isTimeToUpdate** variable as being true which allows the code in **componentDidUpdate()** to fetch the new request from the API and render it. **isTimeToUpdate** is set as being false again at the end of this block of code.

```
//fetches new data for search component, dependent on "isTimeToUpdate" boolean variable
async componentDidUpdate() {
  if (this.state.isTimeToUpdate === true) {
    try {
      const API_URL = this.state.API_URL;
      // Fetch or access the service at the API_URL address
      const response = await fetch(API_URL);
      // wait for the response. When it arrives, store the JSON version
      // of the response in this variable.
      const jsonResult = await response.json();
      if (jsonResult.status === "failure") this.setState({ isData: false });

      // update the state variables correctly.
      this.setState({ apiData: jsonResult.results });
      this.setState({ isFetched: true });
      this.setState({ isTimeToUpdate: false });
    } catch (error) {
      // In the case of an error ...
      this.setState({ isFetched: false });
      // This will be used to display error message.
      this.setState({ errorMsg: error });
      this.setState({ isTimeToUpdate: false });
    }
  }
}
```

FILTERS

We have also implemented filters for these searches (e.g. “Italian”). The **filters array** holds an ID and the name of a “&cuisine” that can essentially be appended onto the end of the API query string for a more specific search.

applyCuisineFilter() is similar to editSearchTerm(event) but slightly more complicated as it being used with a **drop-down menu** and the filters array. To do this, **findCuisineByID()** is used to retrieve the id of the cuisine in the filters array and the matching name is fed into the API query string. We also added “No Filter” as an option to disable the filters.

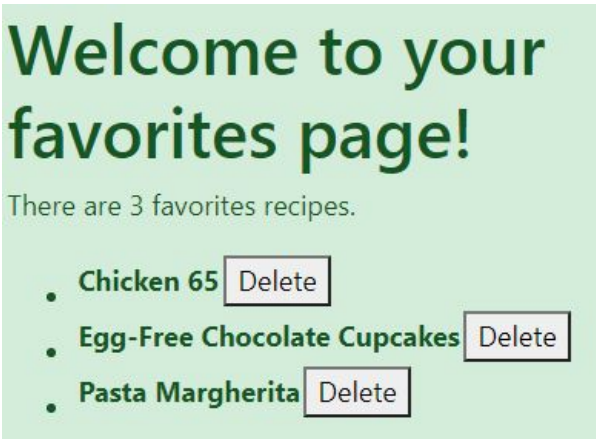
The filters required a lot of debugging to eliminate bugs.

DISPLAYING FAVOURITES PAGE

Based on our design we decided to add a functionality to store the favourite recipe for the user on the app. We tried storing it locally using **npm**. After failing to store locally due to limitations on CodeSandox, we moved to real-time cloud storage developed by Google - **Firebase**. It allowed us to store our JSON file on the cloud server and get it whenever we requested it. We created a config file ‘myFirebase.js’ given by Firebase for the setting up database for the application.

MyFirebase configures and initializes our firebase database. We used **ref.on()** for fetching the **favData** array we need to display and modify favourite recipes. Then we used the **push()** function to push all the updated databases to the cloud and display it with the opening of our Favourites Page.

Below are small snippets of how our data is displayed on our app and how it’s stored in the Firebase server.




```
getMessagesFromDatabase() {
  // referencing the fav array on our Firebase database
  let ref = Firebase.database().ref("fav");

  //mapping data from Database and putting it in newMessagefromDB array
  ref.on("value", (snapshot) => {
    let msgData = snapshot.val();
    //console.log(msgData);
    let newMessagesFromDB = [];
    for (let m in msgData) {
      // create a JSON object version of our object.
      // contained within the database
      let currObject = {
        id: msgData[m].id,
        image: msgData[m].image,
        imageType: msgData[m].imageType,
        title: msgData[m].title
      };

      // add it to our newStateMessages array.
      newMessagesFromDB.push(currObject);
    } // end for loop
    // set state = don't use concat.
    this.setState({ favData: newMessagesFromDB });
    //console.log(newMessagesFromDB);
  }); // end of the on method
} // end of getMessagesFromDatabase()
// What happens when we click on the 'Send' button. The button is only active
// if we have a valid form.
// When we press this button, we would like to send the message to our Firebase database.
```

DELETING FROM FAVOURITES PAGE

As displayed above, we see all our favourite recipes stored on the favourites page. You can delete the recipe by pressing the **Delete** button. Delete button will take the recipeID on button **onClick()** and pass it to **deleteRecipeObject()**. The function uses a **filterByRecipeID()** filter to get the remaining array removing the recipe with the same RecipeID and sets the update to the local favourite array and sets it to cloud DataBase. Then we set favData and replace the real time cloud database with the one we have locally.

```
/** delete the recipe object from user selection */
deleteRecipeObject(RecipeObjectIDToDelete) {
  console.log("deleted");
  // get the current state array dbData holding our data
  const localStockObjects = this.state.favData;

  // apply the filter function to remove the object we wish to delete.
  // This is our dbData array WITHOUT the object for deletion.
  const updatedLocalRecipeObjects = localStockObjects.filter(
    this.filterByRecipeID(RecipeObjectIDToDelete)
  );
  console.log(updatedLocalRecipeObjects);
  // set state in the application
  this.setState({ favData: updatedLocalRecipeObjects });

  // update the firebase database - using set from Firebase API
  // we replace the data at recipeData with the new dbData array (in state)
  // or using our local variable.

  Firebase.database().ref("/fav").set(updatedLocalRecipeObjects);

}
```

ADDING TO FAVOURITES

AddToFavourite() function helps us to add a new recipe to our favourite page. We send the recipe object to **addToFavourite()**. Then we push the recipe object to the **favData** array and then set the **favData** to the online database on Firebase.

```
addToFavorite(r) {
  // create our new message object which will be inserted into the database.
  let newMsgObj = {
    id: r.id,
    image: r.image,
    imageType: r.imageType,
    title: r.title
  };
  console.log(newMsgObj);

  let localMessages = this.state.favData;
  let objId = newMsgObj.id;
  const value = localMessages.some((elem) => elem.id === objId);
  console.log(value);
  //let localMessages = this.state.dbData;
  // add our new object. We can use push here.
  if (!value) {
    localMessages.push(newMsgObj);
    console.log(localMessages);
    /*For basic write operations, you can use set() to save data to a
    specified reference, replacing any existing data at that path
    Using set() overwrites data at the specified location, including any child nodes.*/
    //Firebase.database().ref("/fav").set(localMessages);
    // restore state of this component back to the  default.
    this.setState({ favData: localMessages });
    Firebase.database().ref("/fav").set(localMessages);
  }
}
```

SWITCHING PAGES

When designing the project, we decided that we would first use conditional rendering for page switching and avoid routing unless this didn't work. Fortunately, it worked perfectly.

The page switching functionality relies on three components. First, the functions dealing with the results of clicking on page buttons are implemented in the App.js file. Then, the buttons are displayed by two components. The **Start.js** file deals with the start page (as its disposition is really different from the other ones). For all the other pages, the top bar containing the buttons is managed by an independent component, **Selectpages.js**. We have added the changePage function on **onClick()** function**.**

DISPLAY RECIPE

For displaying a complete recipe of a title, we tried displaying a complete recipe updating the state variable of API_URL, but due to asynchronous function of componentDidUpdate. We were facing a lot of difficulties with the same. So we made a different fetch call for the Show Recipe component. On pressing the '**Show Recipe**' button on the Search page. It sends **recipeID** to the function which passes the ID value to the **ShowRecipe.js** Component. After successfully fetching the json file from the API, we map to display on the showRecipe Component.
While mapping our json file we noticed our array of ingredients where we had to use nested mapping to display that as shown in snippet below

```
<h2>Steps: </h2>
<ol>
  {Data.analyzedInstructions.map((s) =>
    s.steps.map((t) => <li key={t.number}>{t.step}</li>)
  )}
</ol>
```

RANDOM RECIPE

We came up with a fun functionality to give the user a random recipe. The Spoonacular API has already made our work a lot easier by providing a URL to fetch a random recipe out of the database. Then we just had to map that to our component.

SHOPPING LIST

In the start of the app development, we intend to use a shopping list as a way in which a person can view the ingredients they might require to cook all the meals they've planned throughout the week. But having failed to complete the calendar function for our app, we had to drop it. We made a visualisation of the shopping list component and how it will look after if it's functional.

BOOTSTRAP

The last part of our work has been to make our app more aesthetically pleasing. Therefore, we have used Bootstrap. First, we have chosen a picture to use as a background to give our app a visual identity.

After applying the background, we changed the view of the buttons to match them to the theme of our app. We made the display table more readable. We went white text on the app and colored the components to green to make it more readable.

Chapter 4: Overall evaluation

TESTING

We carried out extensive testing of the app to overcome certain issues. Mainly, when the issues with the API keys were causing our code to throw unexpected .map() errors, this forced us to use console.log() throughout the project so we could fully comprehend the issue. As a result, we have a lot of conditions now for **error checking.**

We also let our isolated family and friends that we live with try out the app to gain some outside perspective ("why doesn't your app work like this?").

WHAT WOULD WE ADD AND IMPROVE IF WE HAD MORE TIME

If we had another 4-6 months, there's so much more we would do - what we've achieved in this phase of the project would certainly just be the beginning.

Ideally, we would really like to be able to complete the **Calendar** functionality (almost completed as previously mentioned but hindered by storage issues). It would help the user to plan meals (save recipes) for the specific days, and give an overview of the planned meals in the coming weeks. Once we did this, we could create a proper **Shopping List** (currently waiting to be used by this Calendar function), which would extract ingredients for dates chosen using the date picker.

Letting users **rate and contribute** their own new **recipes** would be a great way to keep users engaged, and let them experiment and be creative. We also like to let users **share recipes** on social media (Kevin has done something similar in a game using the Facebook API before), and leave comments.

For those who like to use our app to improve their diets, displaying the nutritional value would be extremely useful. Furthermore, in our opinion, a **"back" button** on every page would improve usability.

ARE WE HAPPY?

At the end of this project, we are happy with what we've achieved and our workarounds for certain bottlenecks. However, that does not mean that there isn't room for improvement as previously discussed. Some issues have been easier to deal with than others (those irritating API limitations) but all of them have improved our programming skills in React JS.

As a group, we were initially quite awkward as we've never met each other in real life and some initial internet connectivity issues made conversations hard to understand, and meetings harder to organise. However, we overcame these challenges and built a good rapport as a group.

Seeing what the four of us have been able to produce, while not even being in the same country, gives us confidence in what we will be able to do in the not-so-distance brighter future.