|  |  |  |
| --- | --- | --- |
|  | **T.C.**  **MANİSA CELAL BAYAR ÜNİVERSİTESİ**  **MÜHENDİSLİK FAKÜLTESİ**  **BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ** |  |

**Vehicle Transportation-Logistics App**

**Lisans Bitirme Tezi - II**

**HAZIRLAYANLAR**

**ARDA DUMANOĞLU 190315072**

**DANIŞMAN**

**Dr. Öğr. Üyesi Zeynep ÇİPİLOĞLU YILDIZ**

**MANİSA 2022**

|  |  |  |
| --- | --- | --- |
|  | **T.C.**  **MANİSA CELAL BAYAR ÜNİVERSİTESİ**  **MÜHENDİSLİK FAKÜLTESİ**  **BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ** |  |

**Vehicle Transportation-Logistics App**

**Lisans Bitirme Tezi - II**

**HAZIRLAYANLAR**

**ARDA DUMANOĞLU 190315072**

**DANIŞMAN**

**Dr. Öğr. Üyesi Zeynep ÇİPİLOĞLU YILDIZ**

**MANİSA 2022**

**TABLE OF CONTENT**

Page

FIGURE LIST 5

PROBLEM DEFINITION 9

LITERATURE ANALYSIS 10

METHODS AND TECHNOLOGIES TO BE USED 13

PROJECT TEAM AND TASK SHARING 22

REFERENCES 22

INTERDISCIPLINARY STUDY AREA 23

**1. LOGIN** 24

1.1 Login UI 24

1.2 Authentication with API 30

1.3 Auto-Login With Token 34

1.4 Logout 36

**2. SCREEN NAVIGATION** 38

**3. MAIN MENU SCREEN** 39

**4. VEHICLE WAITING FOR UNLOADING SCREEN** 44

4.1 Displaying Vehicle List Pulled From API 44

4.2 Add Vehicle to the List with Pop-Up Modal 50

4.3 Delete Vehicle from the List 54

4.4 Modify Vehicle values with Pop-Up Modal 56

4.5 Add/Delete Details for each Vehicle 58

4.6 DatePicker for displaying vehicles according to the selected date 64

Page

**5. ADD PICTURE** 69

5.1 Take Photo With Camera 70

5.2 Choose Photo From Gallery 74

5.3 Delete Picture 76

**6. MobX for State-Management** 80

CONCLUSION82

**FIGURE LIST**

**LOGIN**

Figure 1.1 Login Screen From Android Emulator

Figure 1.2 Show and Hide Option For Password.

Figure 1.3 Wrong username or password alert.

Figure 1.4 Username or password can not be null warn.

Figure 1.5 Login condition for testing, written in API document.

Figure 1.6 After Successful Login it navigates to the Main Menu Screen.

Figure 1.7 Design part from React-Native. Similar to the CSS.

Figure 1.8 Login Screen return part which we see on the screen.

Figure 1.9 Storing username and password values using the useState hook.

Figure 1.10 InputBoxes return code, the codes we see printed on the screen.

Figure 2.1 LoginPressed function responsible for Login with Authentication.

Figure 2.2 Login Method Calling Rest API using AXIOS library.

Figure 2.3 Response from API after login with wrong username or password.

Figure 2.4 Response from API after Login with valid username or password.

Figure 2.5 SetTokenFunction for storing Token value that returns from API and stored in local storage thanks to the Async Storage function.

Figure 2.6 UseEffect Hook to let us Login after checking the Success value.

Figure 3.1 UseEffect Hook for checking if Token is null or not For Auto-Login.

Figure 3.2 Asking if we want to exit after pressing the HardwareBack Button.

Figure 3.3 Asking if we want to exit after pressing the HardwareBack Button(code version).

Figure 4.1 After Pressing Logout on Menu Screen.

Figure 4.2 After Pressing Logout in Menu Screen(Code Version).

**SCREEN NAVIGATION**

Figure 1.1 Screen Navigator that lets us navigate between each screen.

**MAIN MENU SCREEN**

Figure 1.1 Main Menu Screen returns this code.

Figure 1.2 Updated code that creates vehicle and image list on the Main Menu screen.

Figure 1.3 An array that initially holds a list of images with as many empty arrays as the number of vehicles.

Figure 1.4 Main Menu Screen’s useEffect for getting Vehicles From API in the background.

Figure 1.5 Pulling Data From API and storing locally with this method in Main Menu Screen.

Figure 1.6 Pulling Vehicles data from API with this URL and pulling as JSON array format using Axios library.

Figure 1.7 The JSON Vehicle List data returned From API.

**VEHICLE WAITING FOR UNLOADING SCREEN**

Figure 1.1 Vehicle Waiting For Unloading Screen with the part of data was pulled from the API.

Figure 1.2 We suppose to get data locally where we set async locally in Main Menu Screen.

Figure 1.3 Displayed Vehicles as ListItem and combined them with FlatList with the array.

Figure 1.4 ListItem function that we see as a rectangle box on the Screen.

Figure 1.5 StyleSheet for designing components Like CSS.

Figure 1.6 Displaying all these ListItems with FlatList Component.

Figure 1.7 Alternatively, we can store the data locally instead of the API.

Figure 2.1 Pop-Up Menu on Waiting Vehicle Screen that is activated after pressing the new record button on the screen.

Figure 2.2 NewRecordModifyPopUp head part of the code.

Figure 2.3 The warning message that comes when we do not make any changes and press the record button on the new record screen.

Figure 2.4 NewRecordModifyPopUp rest of the code.

Figure 3.1 The code responsible for the deletion process in the edit screen.

Figure 3.2 Pop-up screen after pressing the delete button on the edit screen.

Figure 4.1 The warning message that comes when we do not make any changes and press the record button on the edit screen.

Figure 4.2 The code that makes the values appear on the edit screen at the beginning and shows the warning message if there are no changes after pressing the save button.

Figure 5.1 The button that is responsible for showing the details on the modify vehicle screen.

Figure 5.2 Add or remove the details screen that is responsible for details for each vehicle.

Figure 5.3 Confirmation message that comes after pressing the delete button.

Figure 5.4 The code that is responsible for the green add detail button.

Figure 5.5 The code that is responsible for the List of the detail messages that are printed on the screen.

Figure 5.6 Delete Detail function that is responsible for when we press delete.

Figure 6.1 The code that is responsible for the date selection filter at the top of the screen.

Figure 6.2 Date picker that shows the date on the screen when we press.

Figure 6.3 No vehicle found on selected date pop-up message.

Figure 6.4 The code that decides whether the vehicles will appear on the screen according to their login times.

Figure 6.5 NoVehicleFound alert that appears on the screen when there are no vehicles found at that selected date.

**ADD PICTURE**

Figure 1.1 Pictures Screen Layout.

Figure 2.1 Take Photo With Camera UI.

Figure 2.2 Edit Photo as crop, rotate, aspect ratio in Camera UI.

Figure 2.3 FlatList for displaying all images that we select or take a photo of.

Figure 2.4 Bottom part of the Screen code that is responsible for taking a photo or choosing from gallery.

Figure 2.5 TakePhotoFromCamera method code.

Figure 2.6 After Taking a Photo the console output.

Figure 3.1 ChoosePhotoFromGallery code.

Figure 3.2 ChoosePhotoFromGallery in application UI.

Figure 4.1 Updated code that creates vehicle and image list on the Main Menu screen.

Figure 4.2 The UseEffect hook, which is triggered by the change of the total\_index\_for\_picture\_list value after the addition or subtraction operations, and controls whether the addition or subtraction is done with add controller state.

Figure 4.3 Confirmation message before deleting the image and the appearance after deleting the image.

Figure 4.4 ImageList is shown in the console after adding and removing images.

Figure 4.5 Updated TakePhotoFromCamera method which adds the image to the beginning of the ImageList array with unshift method.

Figure 4.6 Delete photo method for deleting photos from ImageList.

**MobX for State-Management**

Figure 1.1 MobX integration for state-management in React-Native.

**PROBLEM DEFINITION**

The problem is that it is very important for companies dealing with transportation to control their vehicles, get a list of all vehicles via API, show the information about these vehicles and the products they carry, record them and organize the information about these vehicles. Adding and removing vehicles from their databases, adding information about these vehicles, and adding images from vehicles make their job easier.

The main purpose of this project was to make an application that controls vehicles, makes entry-exit operations of vehicles, allows us to control, record, and edit these vehicles, adding new vehicle registrations and adding-removing pictures of these vehicles. It was created for use in **waste paper** reception areas. For vehicles entering from the scale, selecting a vehicle over the application and ensuring that the information of the scrap papers in this vehicle is integrated into the existing Scrap Paper program.

And by putting all these options in our application, I aimed to create an application for transport companies where they can see and record information about vehicles and products that they carry and let them control their vehicles.

*Keywords: API*

**LITERATURE ANALYSIS**

A literature review was performed in order to:

– determine the theoretical origin of the term “trans­port logistics”;

– identify the most important or most influential stud­ies;

– find potential definitions of the term;

– develop a literature­based definition of transport lo­gistics.

The analysis of literature sources has been carried out and showed that there is no unified transport logistics defined. In order to find out how much this activity is relevant in practice (business), an exploratory survey also was selected using questionnaires. Qualitative exploratory research was carried out in order to find out how the leaders of the transport logistics sector in the Baltic States define the concept of transport logistics.

Since qualitative research was conducted, the aim was to evaluate not just how many respondents responded, but how to get an image – how they understood the concept of “transport logistics”. From all possible answers, similar definitions are grouped and are presented as a result.

When talking about transport logistics, keywords such as “movement of goods”, “physical transfer of goods (from the point of origin to the point of use)”, “physical dis­tribution function”, and “moving goods from one location to another”, and “transfer of cargo” are mentioned most often in scientific literature as well as with practitioners. These keywords mostly include the transport of goods, but others also include words from the fields of planning and management of transport of goods, such as “efficient distribution”, “planning managing and execution of move­ment of goods”, “managing physical flow”, and “managing and optimizing transport flows”. Since transport, among others, also ensures spatial and time utility, keywords such as “optimal delivery times” and “services rendered on time”. In the frame of transport and logistics, other keywords such as “minimal costs”, “high effectiveness and effi­ciency”, “cost-effective organization”, and “minimal human resources” also need to be included. Considering the inconsistencies in the current body of literature and business use, a consistent and common definition of the term “transport logistics” is certainly needed. It must include various aspects and concepts, con­nected with the term, in an inclusive yet adaptive way, so it encompasses all crucial elements and reflects slight differences in defining the scope of the term. With this in mind, the authors propose the following definition: Transport logistics is a crucial part of the supply chain that in its essence organizes, manages, optimizes and ul­timately performs physical distribution of goods and in­formation through the whole upstream and downstream chain in an efficient and effective manner. Its activities and functions go beyond the traditional transport function in that they also include integration with other supply chain functions such as warehousing, accounting, marketing, or customer relations while taking into account the organizational, financial, commercial, and operational aspects of the supply chain as a whole.

– Transport logistics, even though an often-used term, has been shown in this paper to be lacking a com­monly accepted definition in scientific literature as well as on a practitioner level;

– Research identified only two scientific sources which give a definition of transport logistics, and consequently had to broaden the scope to include other

academic resources, 11 altogether;

– There are also many companies that specialize in transport logistics, and in many cases, this involves international activities as the ‘global economy’ grows

and the supply chains grow longer and more com­plex. From the practitioner viewpoint, research in Baltic transport/logistics companies additionally

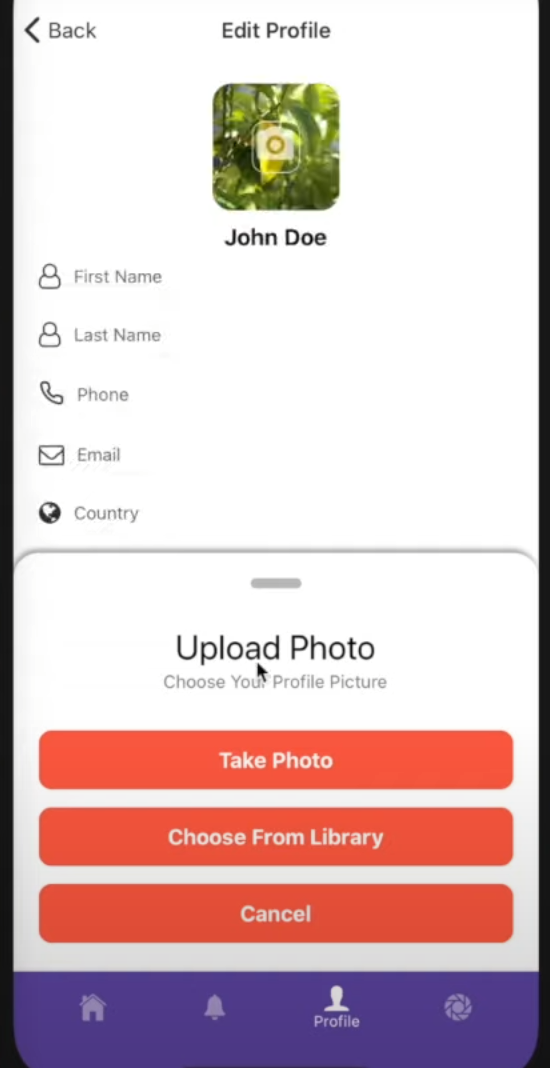
showed that there is no unified definition of trans­port logistics. Not infrequently, this concept is often compared with the transportation process, routing, and cost reduction opportunities;

– Therefore, the authors believe that the work presented in this paper brings a most needed consolidation of scientific, academic, and practitioner viewpoints into a proposed common definition of the term. This will importantly contribute to overcoming the current disparities in using the term in all spheres, mostly in comparison and in relation to logistics and transport alone;

– Further research on the topic should broaden the area where practitioners were surveyed to achieve an even more global consensus, and also try and put the con­cept of transport logistics into a wider framework of supply chain management and logistics**.[13]**

**Source from the app:** [**https://github.com/itzpradip/Food-Finder-React-Native-App**](https://github.com/itzpradip/Food-Finder-React-Native-App)

I was inspired by this app to choose a picture from the gallery and take the photo part. And I got the idea of how to use this image-crop-picker library in react-native.



I inspired the bottom part of this screen. I adapted it to my application. The difference between mine and his is that he is making a profile picture while I print the photos on the screen.

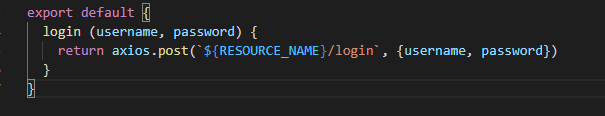
**METHODS AND** **TECHNOLOGIES TO BE USED**

**METHODOLOGY ANALYSIS:**

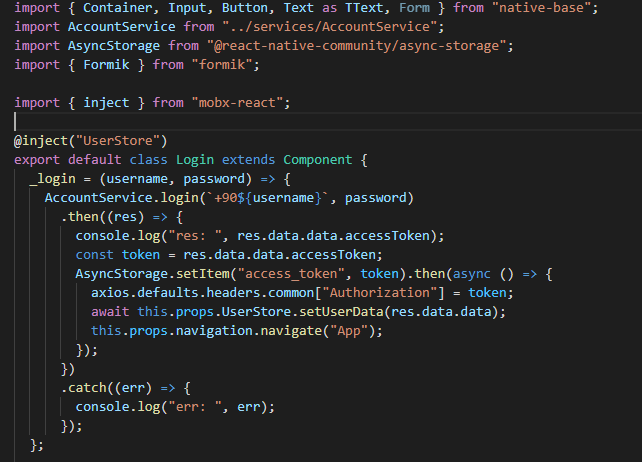
My company gave me access to the mobile applications they developed for giving me ideas and using some parts while developing my project. The names of these projects are "**Aksa Üretim App**", "**SelectAgentMobile**" and “**WatchApp Mobile**”. These projects are similar to my project. I was inspired by the login screen and service connections from them. Now I will analyze the methodologies and shortcomings of their projects and we will talk about are these up-to-date or not. They use BitBucket instead of GitHub due to the unlimited repository.

If we examine service connections they are using **AXIOS[4]** instead of **Fetch API[12]** and when I asked them why. They responded to me and explained that, **if we use fetch API handling JSON data is a 2 step process. First, we must make the request and then call the .json() function on the response since Fetch API sends data with the body property. With Axios, the data is sent through the data property of the options, and it automatically stringifies the data in the response. So, we don’t have to do a lot there as Axios inevitably converts JSON data after the request is resolved.**

And I think this is enough for using AXIOS. Also, we can customize header config, etc. Let's start to examine SelectAgentMobile App first. This code is from their SelectAgentMobile Project.



They said, generally for login or authentication we use the HTTP-post method because of security and vulnerabilities. But my project’s API documentation said us to use the get method for login authentication. So It depends on the project.



They use **MobX[11]** from react. **MobX[11]** is a simple, scalable, and standalone state management library. It is hard to learn and use but also it is very useful for controlling states and it allows us to access all states without sending data between screens. I also used Mobx in my project. They store all the data in the UserStore and they can access it from anywhere. They import it with inject function from mobx. I use **async-storage[6]** for storing data. The disadvantage of async-storage is it is stored without encryption and anyone can access and see that data in device files. I store User Token with Async-storage and it is generated randomly every time therefore that does not give us any vulnerabilities.

They use Formik and Yup for list validation**[5].** They told me to use these. Because if you deal with all inputs and assign them manually you can’t deal and you make it complex. With these, you can take multiple inputs from Form. And this is the perfect solution and all of these solutions are up-to-date. And even react-native says that async-storage is deprecated. But since I developed this project and don’t publish it anywhere it does not make any problem for us for now.



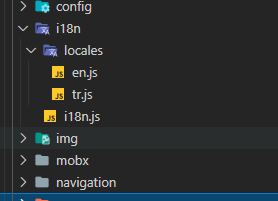
The validation part of my project was done similarly to this one. I talked about the current and beautiful methods they have used so far for now. If we talk about the shortcomings the biggest problem even they admit when I suggest them they still use class components and don’t use

Hooks in react native. React native is evolved to the function components. Function components have too many advantages in react native official documentation they explained well. The most important one is Hooks. Function components are defined with arrow function javascript style. With class components they use **componentWillUnmount**() and **componentDidUpdate()** which is called **component lifecycle methods in react.**

In the modern equation of these lifecycle methods in function components is the **useEffect** hook which has the same functionality and is replaced with lifecycle methods. But the advantage is in class components we can only define one lifecycle method. But in the function component, we can define the unlimited useEffect() method anywhere.

They told me that in their next project, they will examine this project that I made and switch to function components, which are the current modern solution.

In their second project which is the “Aksa Üretim App”. When I examine this project I asked them What is **I18n** means. And they told me that internationalization (i18n) is the process of preparing software so that it can support local languages and cultural settings. For example this code. The texts that will appear on the screen depend on the language pack we choose. I don’t need that so I skip this part. And the rest of them are the same as the SelectAgent Mobile project.



I don’t use their WatchApp Mobile because they are using Google services and calculating watch accuracy with some methods. I only examine it but nothing is different from other projects they all use the same methodologies and technologies in this project as I mentioned earlier. My project is not related to these parts but the rest of them like Login or service connections and the others are done the same as other projects they did.

**TECHNOLOGIES:**

**React-native:**

React Native is a JavaScript framework for writing real, natively rendering mobile applications for iOS and Android(cross-platform). It’s based on React.

**React- Hooks:**

It allows us to use state and other React features without writing a class. Hooks are the functions that "hook into" React state and lifecycle features from function components. It does not work inside classes.

**Javascript:**

JavaScript is a text-based programming language used both on the client-side and server-side that allows us to make web pages interactive. Javascript is used by programmers across the world to create dynamic web content like applications and browsers.

**Visual Studio Code :**

Visual Studio Code is a code editor redefined and optimized for building and debugging modern web, cloud, and mobile applications.

**Rest API:**

A REST API (also known as RESTFUL API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for representational state transfer.

An API is a set of definitions and protocols for building and integrating application software.

**JSON:**

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax. It is commonly used for transmitting data in web applications (e.g., sending some data from the server to the client, so it can be displayed on a web page, or vice versa).

**Axios:**

Axios is a promise-based HTTP client for node.js and the browser**.** Axios is a Javascript library used to make HTTP requests from the node. js or XMLHttpRequests from the browser and it supports the Promise API that is native to JS ES6. It can be used to intercept HTTP requests and responses and enables client-side protection against XSRF. It also has the ability to cancel requests.

**Android Emulator:**

The Android Emulator simulates Android devices on our computer so that we can test our application on a variety of devices and Android API levels without needing to have each physical device. The emulator provides almost all of the capabilities of a real Android device.

**React Native CLI:**

React Native CLI is a built-in feature that helps us take control over the management of the project locally. We can create and run our applications. We can create a project by simply using this command.

**Formik Validation:**

Formik is designed to manage forms with complex validation with ease. Formik supports synchronous and asynchronous form-level and field-level validation. Furthermore, it comes with baked-in support for schema-based form-level validation through Yup.

**Async Storage:**

AsyncStorage is an unencrypted, asynchronous, persistent, key-value storage system that is global to the app. It should be used instead of LocalStorage. It is recommended that you use an abstraction on top of AsyncStorage instead of AsyncStorage directly for anything more than light usage since it operates globally.

**NPM:** NPM is the package manager for the Node JavaScript platform. It puts modules in place so that node can find them, and manages dependency conflicts intelligently. It is extremely configurable to support a wide variety of use cases. Most commonly, it is used to publish, discover, install, and develop node programs.

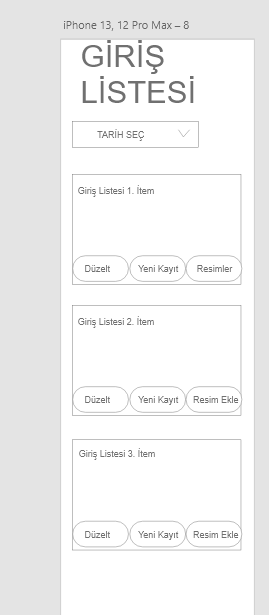
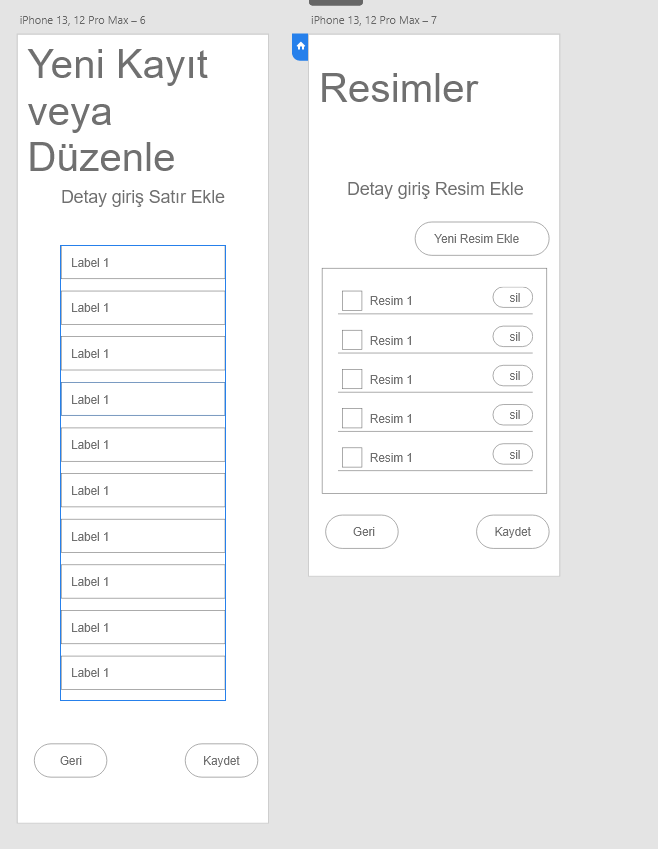
**React Native Libraries:**



I am using these libraries from react-native.

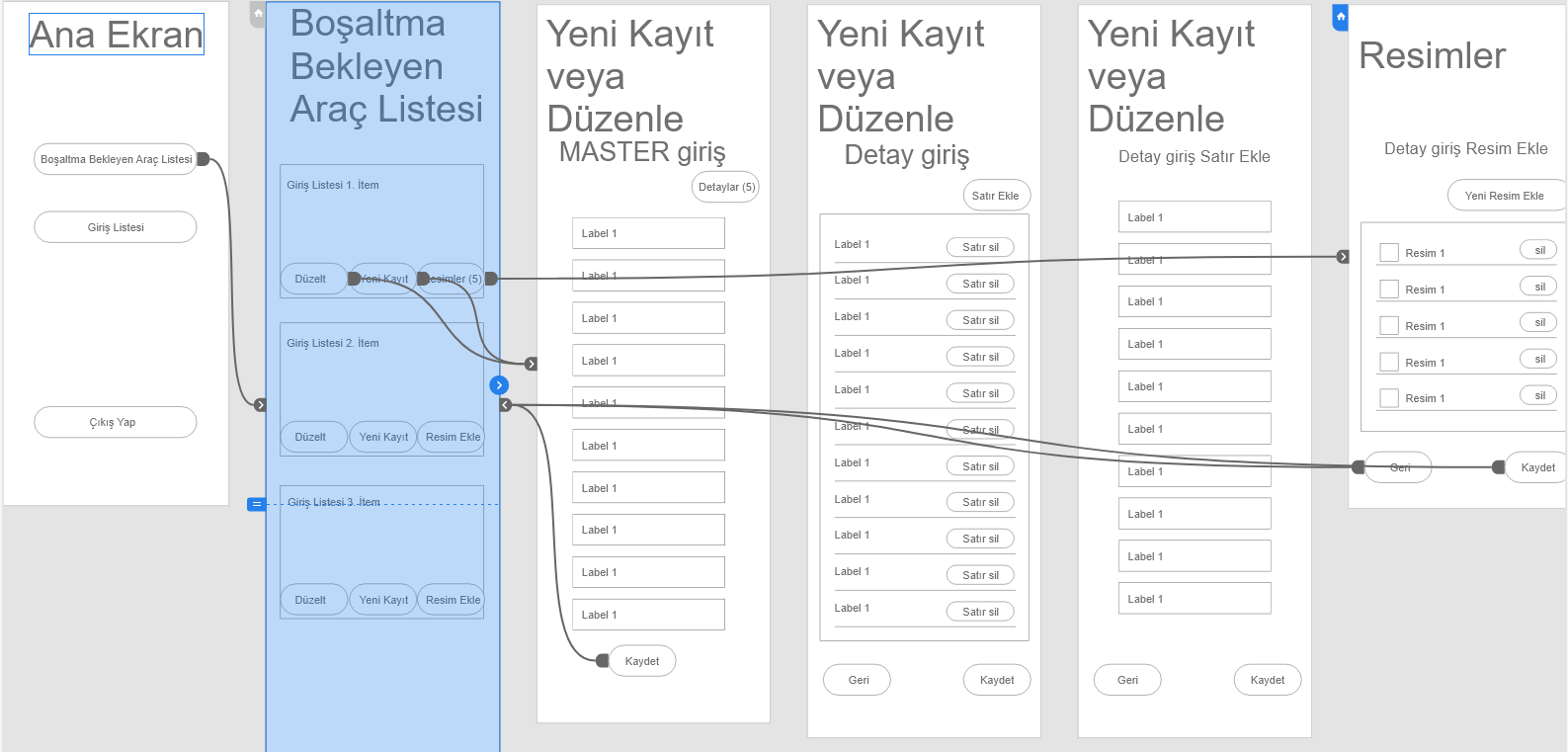
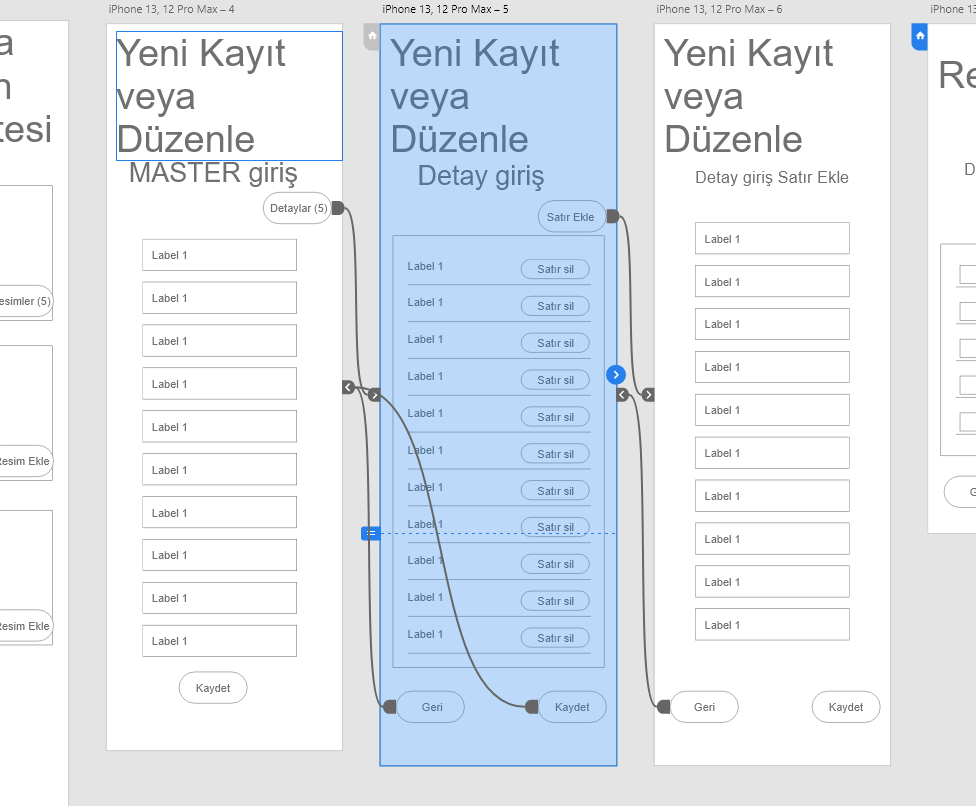
**WIREFRAME DESIGN FOR SOLUTION**



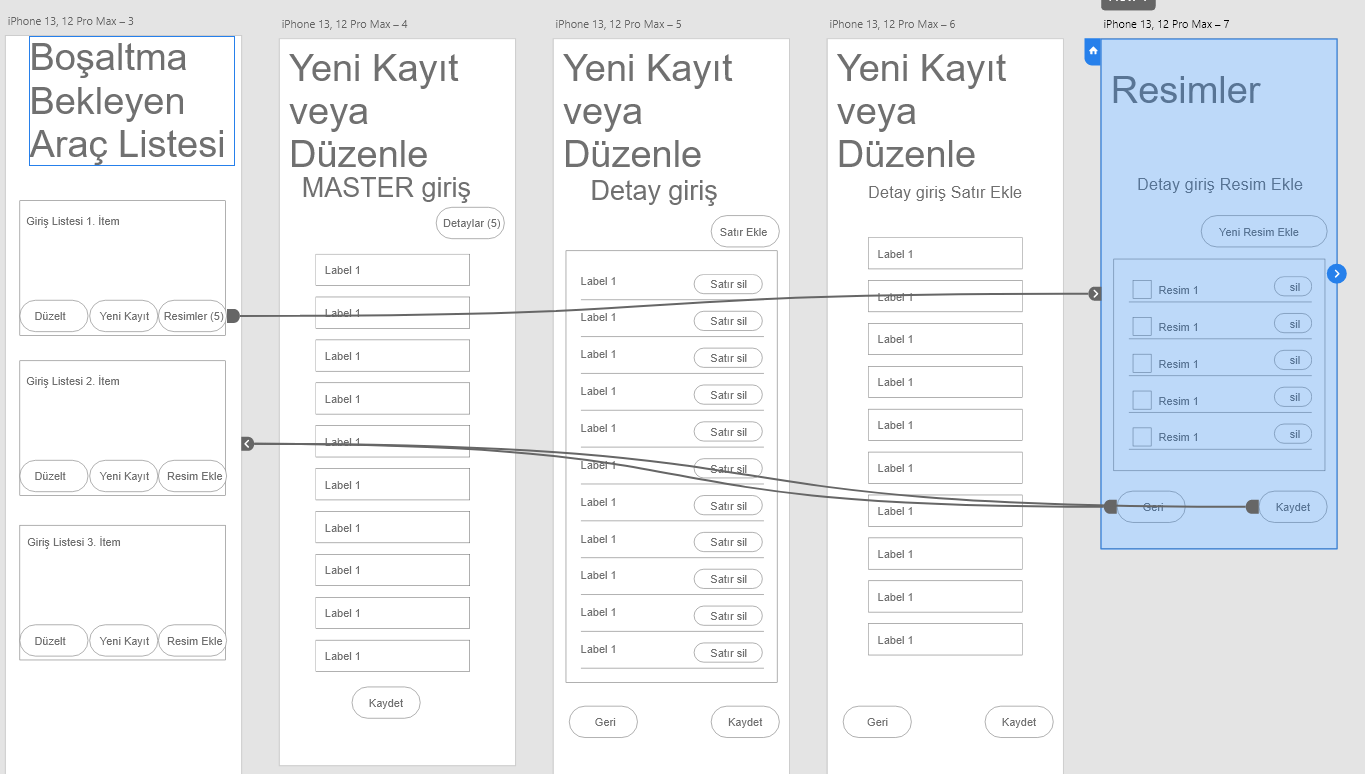


I made these designs through the adobe xd program. These are sketchy drawings that I used to realize the idea I had in mind at the moment, which is imprecise as it is a wireframe design. This design is simply drawn to give me an idea. I am still developing this application. At the last moment instead of the new record or modify screen for modifying values, I turned to the design that appears in the form of a pop-up modal when I press the “düzelt” button in the vehicles waiting for unloading screen and it allows us to change the values.

I also designed the **PROTOTYPE** parts that show when we press something and what it does.

As I mentioned earlier these are prototypes and the design is a wireframe design. Therefore nothing is imprecise it is just a rough sketch.

**PROJECT TEAM AND TASK SHARING**

I developed this project by myself, also my advisor always gave advice and told me what to add. And it helped me to design the structures and interface that I need. He also examine my code and gave some advice for writing clean code or defining variable names. He explained to me how to get data from API or how to make service connections, how to authenticate with the token for the login screen, how to use token for auto-login, and formik validations for taking multi inputs from the user for edit, and new record screen, etc. And also, my company has given me access to similar apps(which I explained in the methodology analysis part) they have developed before for examining these projects and applying some parts to my project.

**REFERENCES**

**[1]** <https://reactnative.dev/docs/getting-started>

**[2]** <https://tr.reactjs.org/docs/hooks-intro.html>

**[3]** <https://www.w3schools.com/js/>

**[4]** <https://axios-http.com/docs/api_intro>

**[5]** <https://formik.org/docs/guides/react-native>

**[6]** <https://reactnative.dev/docs/asyncstorage>

**[7]** <https://reactnavigation.org/docs/getting-started/>

**[8]** <https://github.com/ivpusic/react-native-image-crop-picker>

**[9]** <https://nativebase.io/>

**[10]** <https://www.researchgate.net/publication/228792450_Interdisciplinary_modeling_for_logistics_design>

[11] <https://mobx.js.org/README.html>

[12] <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API>

[13] <https://journals.vilniustech.lt/index.php/Transport/article/view/6965/6106>

**INTERDISCIPLINARY STUDY AREA**

* **Logistics / Transport / Transportation**

The application I developed is directly related to this section.

The design process of a logistic system requires a **soft systems approach** to deal with different perceptions. The **soft systems approaches** consider a system a subjective perception: dependent on the observer the same system is presented in different ways. The observer himself may also be part of the system and may have his objectives besides the system’s objective. Soft systems approach therefore are mainly aimed at the understanding and the formulation of these so-called ill-defined problems and addressing the “what” question instead of the “how” question.

The design process starts with a so-called ill-defined problem. The first steps of the process must lead to an agreement on the objectives and conditions. By then it is called a well-defined problem. Using hard systems approach only would pass over the proper objective definition and will lead to:

- Accepting system boundaries as given. For example, looking at the effect of economic lot sizes, if one does not take the environment of the total supply chain into account, the savings may be smaller than the extra costs.

- Considering elements as being naturally defined. If one regards an organization as a system, often the existing departments are regarded as the elements. But the departments are the result of design processes in the past. By doing so, the assumptions and starting points of these earlier design processes are implicitly imported into the new design process with its new objectives and in a changing environment.

Now the problem is to find a system concept in a hard systems approach, which can be generally applied within the design process of a logistic system, taking the soft systems approach into account, and which can form a more or less lasting framework for specification and review of logistic systems.

Such a system concept will be called a **conceptual system model**. Only a small number of such conceptual models have been defined. Checkland [1981] positions the use of these models in his Soft Systems Methodology (SSM), the most widely used and accepted soft systems approach.**[10]**

**1. LOGIN**

**1.1 Login UI**

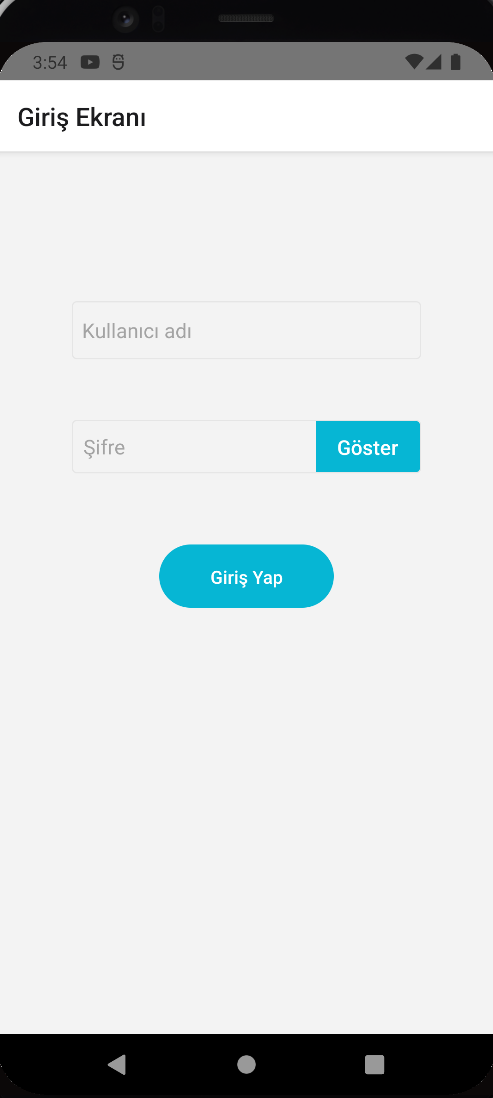


Figure 1.1 Login Screen From Android Emulator.

The first page welcomes us to log in to the application with a username and password.

It allows us to check the values via the Rest API and lets us log in if it's successful (Authentication). We can change the visibility of the password by clicking the show and hide option.

****

Figure 1.2 Show and Hide Option For Password.

If the username or password is wrong it gives us an error.

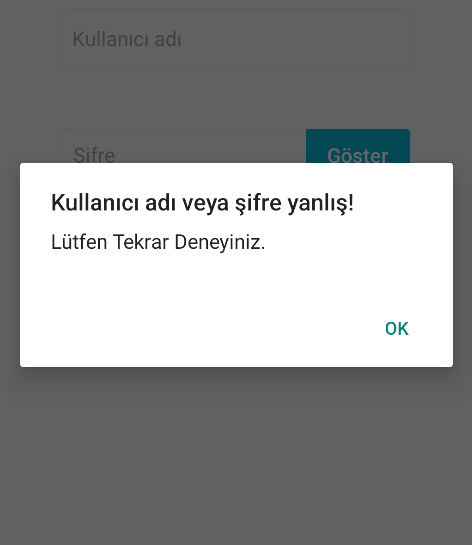
****

Figure 1.3 Wrong username or password alert.

Also if one of them is null it warns us.

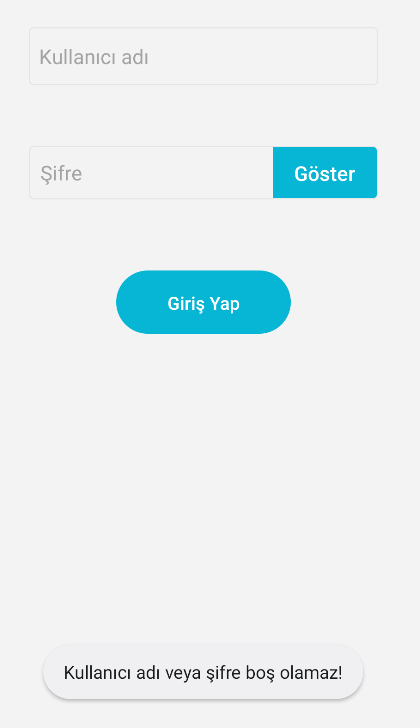
****

Figure 1.4 Username or password can not be null warn.

For now, they told us(Rest API owner Company) that we can log in by typing **username**: **dx** and **password**: **dx** for testing purposes which is written in their API documentation.



Figure 1.5 Login condition for testing, written in API document

After successful login, it navigates us to Main Menu Screen with login success feedback.

****

Figure 1.6 After Successful Login it navigates to the Main Menu Screen.

Now I will explain the code side of the parts I did. First of all, I want to talk about the design part. It is very similar to CSS. We create a Stylesheet then we set some props and values. We call them like CSS. Also, I add **hidekeyboard** functionality so that when we press somewhere outside of the input parts the keyboard disappears.

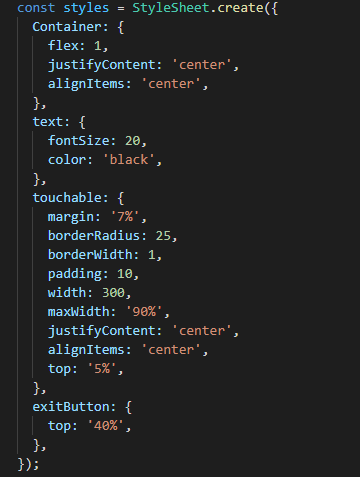
****

Figure 1.7 Design part from React-Native. Similar to the CSS.

For the code side first, we return to the **InputBoxes** custom component made by myself which we see on TextInputs and Login button on the screen. NativeBaseProvider is used because it's required for using native base library components which is in this case Input component in the InputBoxes component which we will talk about later. **HideKeyboard** is a function written by me that whenever we press something outside of inputs it makes the keyboard disappear.

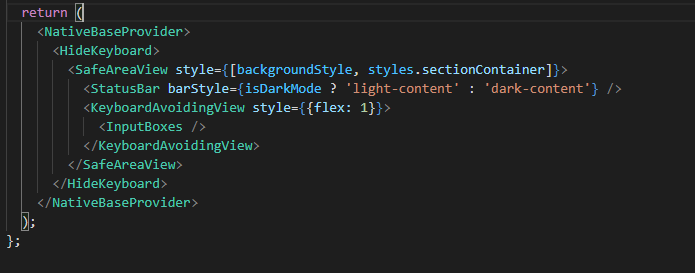


Figure 1.8 Login Screen return part which we see on the screen.

In the InputBoxes component, we use the useState hook From React.js. In earlier versions of react-native developers used class components and state logic. Since the use of hooks was more up-to-date and efficient in later times, class components were replaced by function components. We use useState and useEffect hooks from React. The React useState Hook allows us to track state in a function component. For example, the First userName is actually like a getter method and the second one is the setter method. In this case, we set ‘’ empty as a default value. Later when we say setuserName(“Something”) the username value holds Something as we assign.

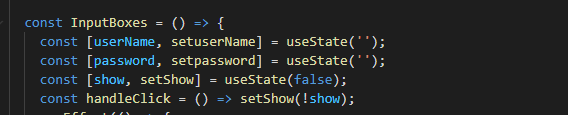


Figure 1.9 Storing username and password values using the useState hook.

Below the code here is the return of the InputBoxes Component which we call earlier. And the codes here are the basis of the codes we see printed on the screen. Stack and Input is a component that comes with the native base library which I imported into the react-native. As we can see when the value is changed OnChangeText props are activated and set our username to the current text that we write on the screen. Also, we hold show value when we click it changes between ‘göster ‘and ‘gizle’. Also, the state is changed as text or password when we examine the code for the password after pressing.

And we have a button on the bottom side. When we press that button it calls the **LoginPressed()** function. So now with that function, we can talk about the second part which is Authentication.

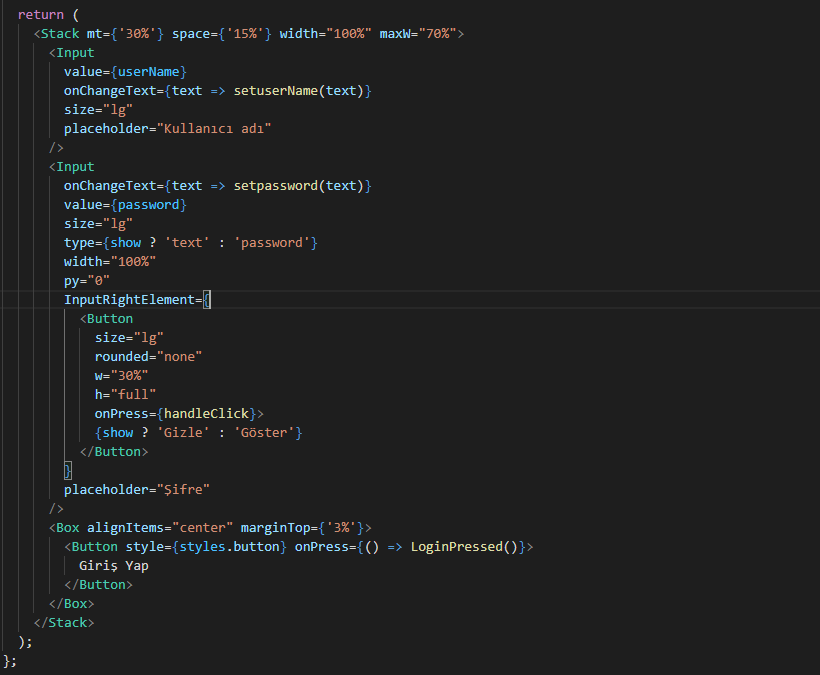
****

Figure 1.10 InputBoxes return code, the codes we see printed on the screen.

**1.2 Authentication with API**

For that part, I want to explain The Authentication with API and the UserToken that returns from the API. We use UserToken for Login and Other processes which are provided by API. First, let's look at the LoginPressed Method.

First, we made a comparison that the username and password are not empty. If it's not null we call the login method from the AccountService.js file which I import there. I will explain the LoginPressed function in detail later.



Figure 2.1 LoginPressed function responsible for Login with Authentication.

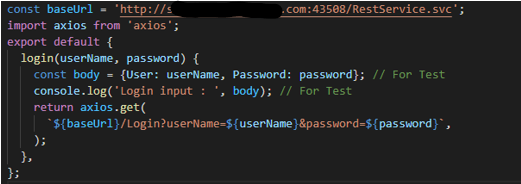
****

Figure 2.2 Login Method Calling Rest API using AXIOS library.

That’s The login method from Account Service.js and it calls Rest API using the AXIOS library. Then the API returns a response that depends on our username and password.

In Login, we put 2 parameters, and it's our username and password values that we type in the text field for login. baseUrl variable takes the service URL from which we pull the data. We use the HTTP-get method for pulling data. We can’t use the post method because we don’t want to access and modify or change any data for that API. We only use the get method for pulling the data from API. Lets going back to the LoginPressed() function. After controlling with if conditions whether its null or not. We call the login function with our username and password parameters. We put then because in Javascript **.then()** method used to deal with **asynchronous** tasks such as our case API call. So it's like an async function with using **await**, that's similar. **.then()** means that it must be executed and wait for API call and returns a response with an arrow function. So we can call response.**data** or response.**status** but in our case, we need data or if we want to access specific data we can call response.data.Token for only getting Token for example. We print the response.data value with console.log here is the result after login in with the wrong username and password.



Figure 2.3 Response from API after login with **wrong** username or password.

Success is false and token is null as expected. We authenticate with these 2 values. But what if we log in with a **valid** username and password**.**

****

Figure 2.4 Response from API after Login with **valid** username or password.

It returns us token and returns success as true. So let's look at back our code again. We authenticate by using whether the success is true or not due to our inputs. If it's true we set Login success true so after we can navigate with using this value in useEffect() and call SetTokenFunction which stores our Token value for later use. If it is false we alert the wrong input or password and setloginsuccess false. Later we use the getItem function from AsyncStorage for taking our ‘UserToken’ for pulling data from API as JSON format.

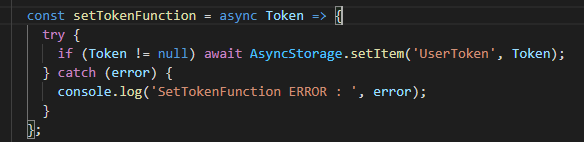


Figure 2.5 SetTokenFunction for storing Token value that returns from API and stored in local storage thanks to the Async Storage function.

And After that, we call setLoginPressedCaller for activating the UseEffect() hook that comes From React. It allows us to perform side effects in our components. It is called after every render as a side effect. When we call useEffect, we’re telling React to run our “effect” function after flushing changes to the React DOM. To summarize when we set this value whether true or false doesn’t matter. It calls UseEffect after every Login press to check if the success is true or not. Finally, If it's true we can go to the Main Menu Screen.

Inside of the useEffect hook first, we checked if the LoginSuccess is true or not which is returned from API. If it's false it alerts as wrong input. If it's true. We can log in successfully and navigate to the Main Menu Screen. Also, we set username and password null if we press the Logout button on Main Menu Screen. Also, we set login success as false if we log out on Main Menu Screen. It navigates us to this Login Screen again. So these are for security reasons. And The LoginPressedCaller is optional. It is called after the LoginPressedCaller value is changed which we set true or false continuously after LoginPress.

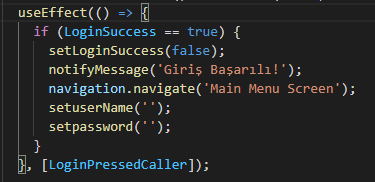


Figure 2.6 UseEffect Hook to let us Login after checking the Success value.

**1.3 Auto-Login With Token**

After Login Successfully. We don’t need to log in every time like Instagram or Youtube App. We log in once then we can use our app. We only log in if we Logout our account. That’s provided by UserToken.

So if we set [] empty in UseEffect that means it is called only once after the first render. We call the getToken Method after when we enter the application every time. **(Figure 2.7)**

Also, We checked BackAction **(Figure 2.8)** if we press hardware back(the leftmost button at the bottom of the screen) It asks us Are you sure you want to exit, and if we hit yes the application closes but remains open in the background.

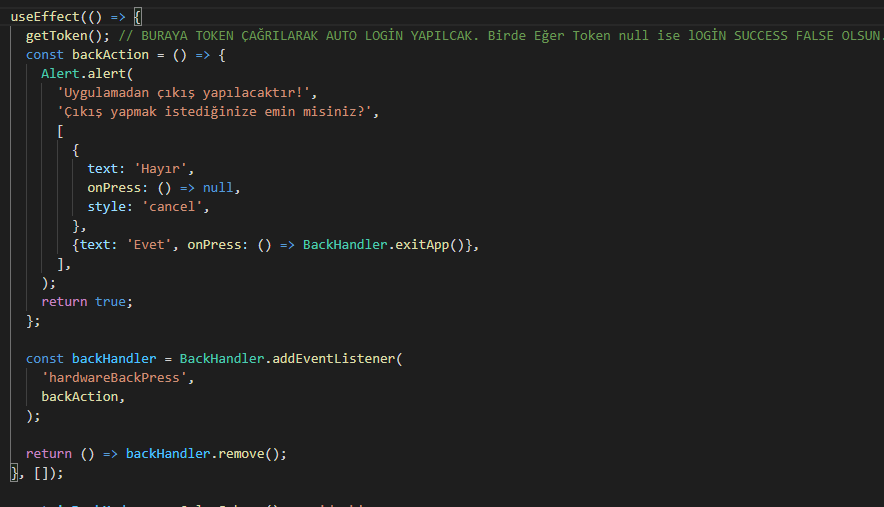


Figure 3.1 UseEffect Hook for checking if Token is null or not For Auto-Login.

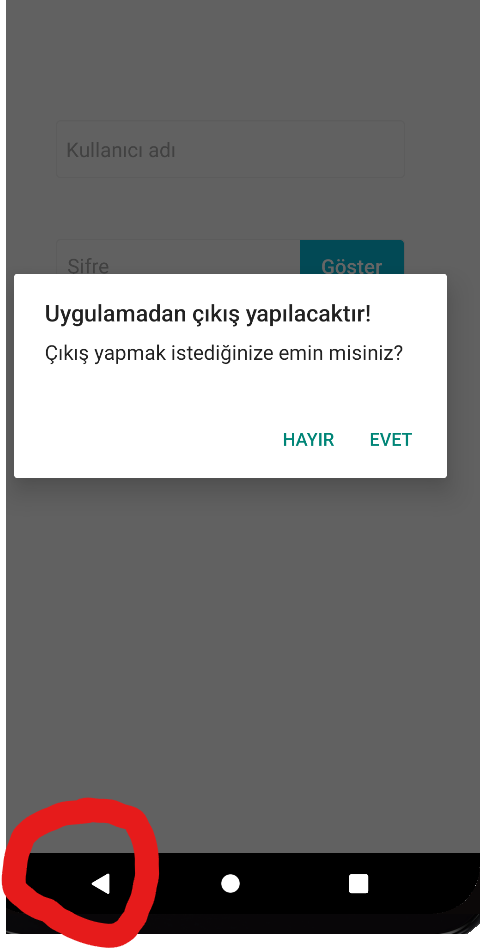
****

Figure 3.2 Asking if we want to exit after pressing the HardwareBack Button.

In the getToken method that we called in useEffect() in **(Figure 2.7)**.It checks if the user token is null or not. If it's not that means we have token and we logged before successfully. We store it in local storage in a device with AsyncStorage.setItem so there is not likely to be lost. We check with AsyncStorage.getItem(“UserToken”) method then it returns a value which is our Tokens value. If it's null we don’t let Login as assigning LoginSuccess = false. Otherwise, we navigate to the Main Menu Screen. It is called after every application is entered.

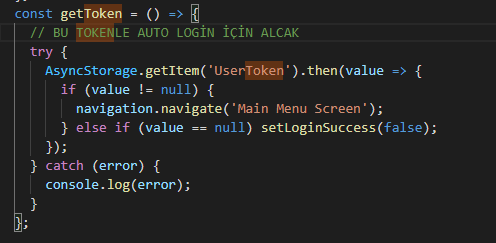


Figure 3.3 Asking if we want to exit after pressing the HardwareBack Button(code version).

**1.4 Logout**

After Navigating Main Menu Screen There is a button Called Logout(Çıkış yap). If we press that button it asks us are sure about logging out and returning to the Login Screen. After We press Yes it returns to the Login Screen. Also for preventing Auto-Login We delete our UserToken by using AsyncStorage.RemoveItem() method and navigate to the Login Screen. So we have to log in with our username and password now. Since we delete our token our Gettoken method returns null from UseEffect and we keep remaining on the login screen.

****

Figure 4.1 After Pressing Logout on Menu Screen.

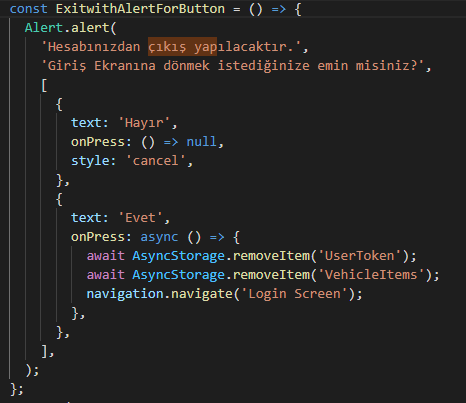


Figure 4.2 After Pressing Logout in Menu Screen(Code Version).

**2. SCREEN NAVIGATION**

In the below code we create every screen and stack them together with the stack navigator. We set the initial route name as Login Screen. We set the name property for navigating between each screen and it doesn't show up anywhere on the screen. We set components as screens that we want to display and it's imported from other javascript files. These components are called and visible as pages after each navigation. We set options property as shown in the title that appears on every screen’s header. We use {navigation} prop on every component’s parameter. Therefore if we want to navigate we can call it navigation.navigate(“Main Menu Screen”).



Figure 1.1 Screen Navigator that lets us navigate between each screen.

**3. MAIN MENU SCREEN**

In the below code we use the View component and call Container inside of Stylesheet. View component works like div in CSS. We display the header name as Ana Menu. Then we have a touchable opacity, it is like a button with feedback but we can customize it inside. When we press The first Button(Boşaltma Bekleyen Araç Listesi on Screen) it calls Onpress property and that navigates us Vehicle Waiting For Unloading Screen as expected. Same as the second one navigates us LoginList Screen. And also we have the Logout Button we talked about earlier. And we have useEffect for getting vehicles from API in the background. The reason I’m calling this right now is we have to display all Vehicles on Vehicle Waiting for unloading Screen. If I call this on Vehicle Waiting Screen it doesn’t print at first because useEffect doesn’t execute before render. It waits first render to execute. To prevent that we must pull JSON data from REST API in this Screen then we store it with asyncStorage as local. After navigating Vehicle Waiting Screen we only need to get this with Async Storage as we stored earlier. That’s the best solution for this case.

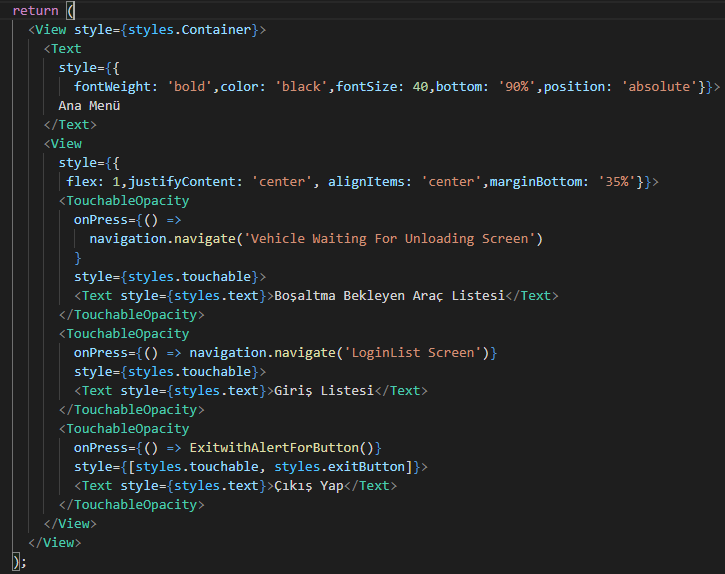


Figure 1.1 Main Menu Screen returns this code.

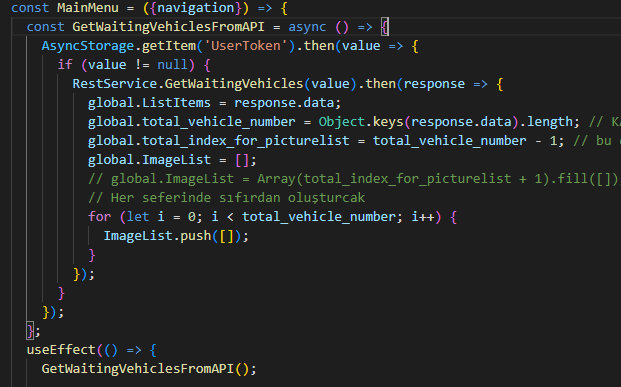


Figure 1.2 Updated code that creates vehicle and image list on the Main Menu screen.

In this photo above, there is the code that I created before and edited now. First of all, instead of assigning the list of vehicles that come from the API one by one with a for loop, I assign directly with a response.data to the List Items. Then I create a global variable total\_index\_for\_picture\_list which is responsible for generating a list of image indexes based on the total number of vehicles. I also created an image list that is responsible for displaying images in the adding image screen. The size of the ImageList is equal to the number of vehicles. And it adds as many arrays with the push method as the number of vehicles. And while adding/removing vehicles, we also add and delete from this Imagelist. In summary, we are trying to synchronize two arrays with each other. The reason we push an array into the array is we may require more than a photo for each vehicle. For my situation first, it adds 21 empty arrays with for method. As we see in the next photo, if we add or delete the vehicle from the Vehicle list, ImageList also adds or deletes for that specific index.



Figure 1.3 An array that initially holds a list of images with as many empty arrays as the number of vehicles.

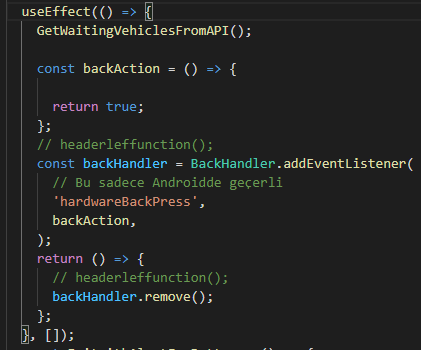
****

Figure 1.4 Main Menu Screen’s useEffect for getting Vehicles From API in the background.

We call **GetWaitingVehiclesFromAPI()** method after first rendering only once in Main Menu Screen I explained the reason on the previous page. I also disable hardwarebackpress on this screen to prevent going back to the login screen without pressing logout. If we look at the GetWaitingVehiclesFromAPI function the below, since we log in successfully by accessing this screen we must need a token as we talked about earlier in the authentication part. First I checked if the token is null, it can’t be null but to guarantee our work, since there may be a system error, I still checked here if it is null or not, and as we expected it is not null. We Call GetWaitingVehicles this is the Second API that we used in this project for Pulling data, the first one is for login and the second one is calling from the RestService.js file. We put token as input because it requires token for accessing data from API. Then it returns a response as we did that before we get response.data and store this as local. And we convert them to String because It is JSON format. It returns a JSON array with JSON objects. Since it looks obsolete to me I updated this code as we see in **Figure 1.2.**

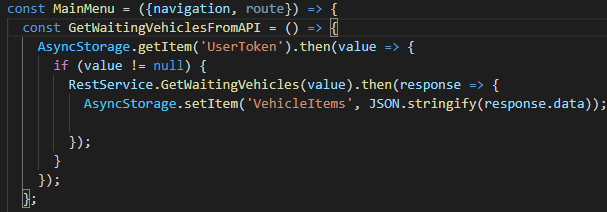


Figure 1.5 Pulling Data From API and storing locally with this method in Main Menu Screen.

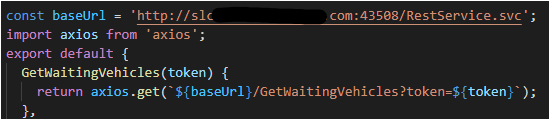
****

Figure 1.6 Pulling Vehicles data from API with this URL and pulling as JSON array format using Axios library.

We are pulling data from this URI template as they told us in their API documentation. It returns data as we can see below which is the response.data.

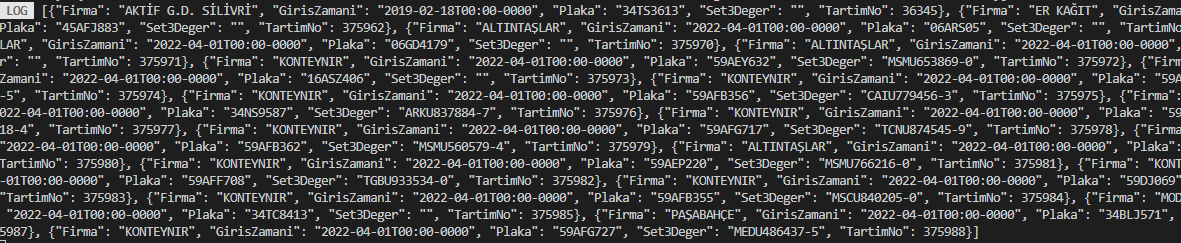


Figure 1.7 The JSON Vehicle List data returned From API.

After that, we only need to get this data since we stored with locally thanks to the asyncStorage and will display them with the Flatlist component, which is the core component in react native, in the Vehicle Waiting Screen. But we can’t use useEffect() to get this because it must be printed on the first render. Unfortunately, useEffect() is called after the first render or each render as we specify. We will talk about this in the next session.**4. VEHICLE WAITING FOR UNLOADING SCREEN**

**4.1 Displaying Vehicle List Pulled From API**

****

Figure 1.1 Vehicle Waiting For Unloading Screen with the part of data was pulled from the API.

In the above code, we define some usestate hooks. ModalVisible state is used to determine whether the adjust value pop-up modal is visible or not. FlatListRenderer is used to re-render the flatlist again if we push something new to the array and it will display thanks to the flatlist component. As I mentioned earlier. We call useEffect only once after the first render to get vehicle list data from API. UpdateVehicles method checks if the token is null or not. It can’t be null but we want to guarantee it. If we have a token then we call API by giving the token value as input. Then it returns some response. Response.data is the vehicle list that we want.

First, we use Object.keys to determine the length of the vehicles. Then we iterate with for loop and push to the array all the values. Then we store these values locally with async-storage.

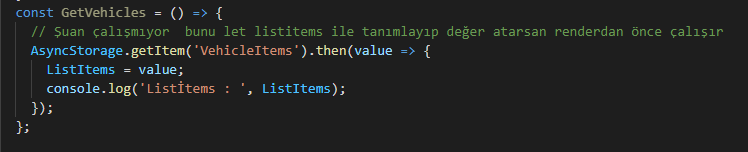


Figure 1.2 We suppose to get data locally where we set async locally in Main Menu Screen.

The problem is we want to assign ListItems before the first render but unfortunately, we can’t call this in the first render. We can only call after the first render in useEffect() and if we do that it is not displayed due to the empty array in the first render after the second render it shows up. Therefore I manually pull all the vehicle data from API. I think that’s the problem with the flatlist component in react-native. So there is no problem. And ironically if I delete the setItem method in Main Menu Screen’s useEffect(). Other solution doesn’t work too. When I ask my supervisor he said that It is not my fault.

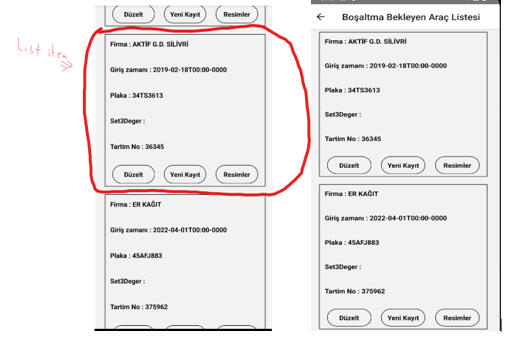


Figure 1.3 Displayed Vehicles as ListItem and combined them with FlatList with the array.

So let’s continue to explain our code. So as we see below we have a ListItem function. This function takes an item as a parameter and returns some JSX. This code is the rectangle boxes of the array that we printed to the screen with the flatlist that we can see in the previous image. We print ListItems with Flatlist which I explain later. It looks like HTML and CSS. It is similar to them. I also print the stylesheet later. If we look at the Text components we see some names like Firma: Plaka: and also {item.something} this is called props. So it is like a variable, we tell that we want to define these values later. After we call these ListItems we have to set these props which are (LoginTime, WeighingNo, Company, etc.) values in the FlatList section. We see some buttons here, the first one is called “düzelt”. When we press that it navigates us to modify the screen.

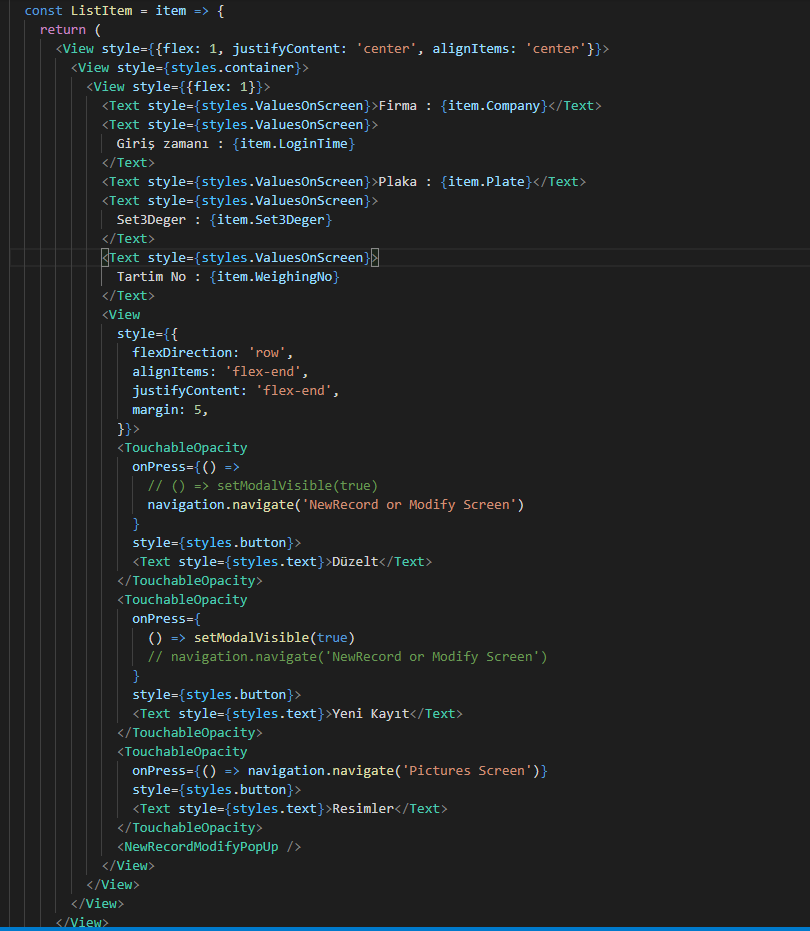
****

Figure 1.4 ListItem function that we see as a rectangle box on the Screen.

Second button is “Yeni Kayıt”. When we press that button Onpress is called and it sets the modelvisible variable to true so it shows up. Therefore our <NewRecordModifyPopUp> component at the bottom pops up on the screen. We will talk about that pop-up for the new record in the next session. And the last one we have “Resimler”. It is basically when we press that button it navigates us to that responsible screen which is Pictures Screen. In the below image this is the all styles for the screen layout. justify-content aligns the main axis(in default it is a column). AlignItems aligns cross-axis(in default it is a row) unless we change flex-direction to the row. (Column is the default).

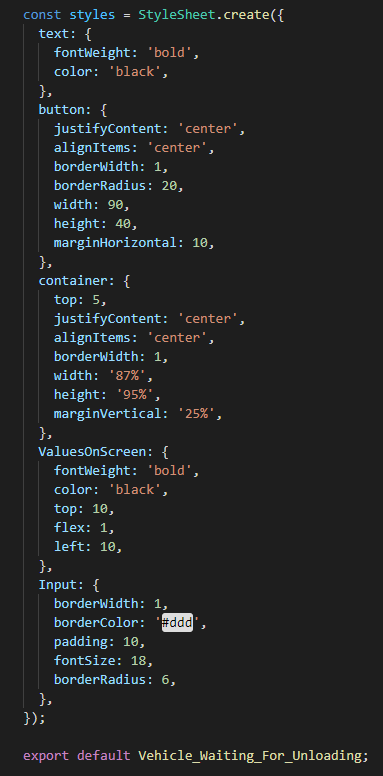
****

Figure 1.5 StyleSheet for designing components Like CSS.

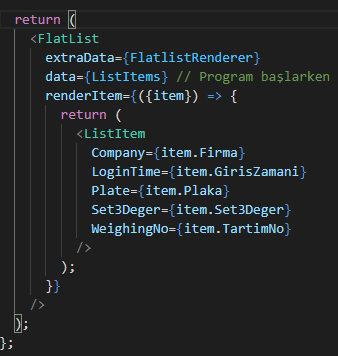


Figure 1.6 Displaying all these ListItems with FlatList Component.

This is the FlatList component that takes data as an array and prints them according to the renderItem condition. In renderitem, we set item prop. This item prop defines the actual item in the data prop in this case it is ListItems. When we add something via pushing that ListItems it is added to the end of the array. ExtraData is an optional prop. It takes the variable FlatListRenderer after when we change this variable’s value like between true-false, it re-renders FlatList again so we can see changes immediately on the screen without waiting. In renderItem, we return ListItem which is a rectangle box on the screen. We print all of the boxes until the array is finished. As I said earlier we set {item.WeighingNo etc.} now it is time to assign these values so they can be shown up on the screen. And the values that we want to print are inside of the ListItems array that is pulled from API. Basically ListItems array returns us JSON objects for example : {"Firma": "AKTİF G.D. SİLİVRİ", "GirisZamani": "2019-02-18T00:00-0000", etc.. We have to get these values via {item.Firma} in this case it is Aktif G.D Silivri and we assign these values so they can show up on the screen. This is all we need, after automating these processes it prints all of them on the screen.

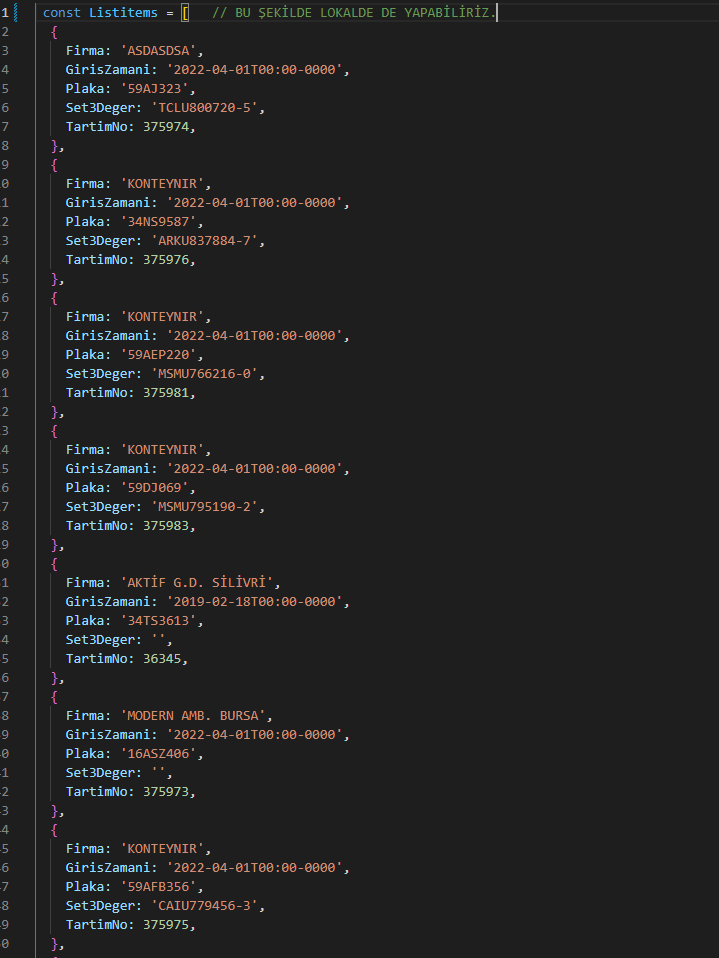
****

Figure 1.7 Alternatively, we can store the data locally instead of the API.

**4.2 Add Vehicle to the List with Pop-Up Modal**

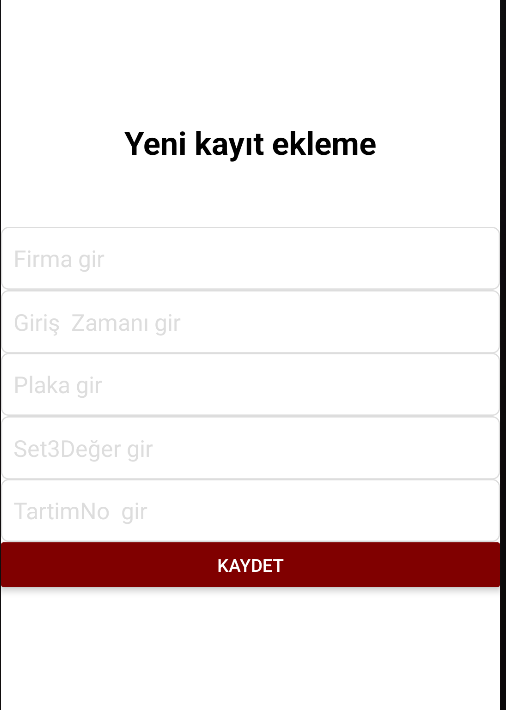
****

Figure 2.1 Pop-Up Menu on Waiting Vehicle Screen that is activated after pressing the new record button on the screen.

When we press the “Yeni Kayıt” Button on the screen this pop-up Modal shows up. After pressing Kaydet Button it adds all the values on the screen and combines all of them as JSON Object then it adds to the end of the array via the push method. Therefore we can display a new element on the screen. NewRecordOrModifyPopup component is responsible for all this.

This method is too long so I divide them and explain them part by part.

So as we see in the image which is **Figure 2.2 first** we use the **Modal** component this component is used for pop-up screens and it is used to set the visibility of some components changing from state to state. We set transparent prop false so we don’t want to see background when it shows up. The visible prop is the most important factor here. It changes when we press something. For example, when we press kaydet it sets as false. So modal is not visible anymore until we press the “Yeni kayıt” button that sets modalvisible true.

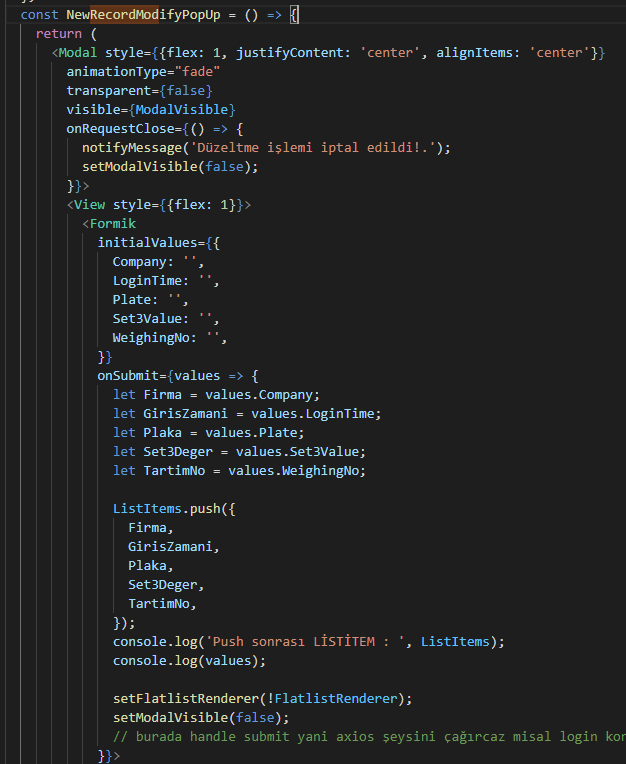
****

Figure 2.2 NewRecordModifyPopUp head part of the code.

OnrequestClose is the prop that controls what happened if we close that with hardware back buttons on the device. So basically it gives us a message and sets the modal’s visibility false. And finally, as I mentioned earlier. We use Formik for multiple inputs with form validation. We set Initial values as null. We have the OnSubmit function and I explain them later. So let's jump to the rest of this code in **Figure 2.3**. We create 2 curly brackets this is the solution that is used for Formik. Inside of these curly brackets. We have 3 methods that come with Formik. HandleChange is responsible for changing initial values with the inputs we write on the screen. HandleSubmit is a callback function that calls Onsubmit when it is called. Values are our form values. We have TextInput Components and onChangeText we update the values with handleChange whenever we write something on the form. On the value part, we separate all of them as value.something . So after we fill the form when we press save or “Kaydet” on the screen. These values are going to the OnSubmit function as a value parameter. On the bottom part, we have a button that calls onsubmit.

Now we can talk about the onsubmit function. We take all these values and assign them to the variables. Then we combine all these variables as a JSON object and push it into the ListItems array. After that, we call FlatListRenderer for activating the Extradata prop on Flatlist which is responsible for re-rendering immediately on the list so we can see our values on the screen.

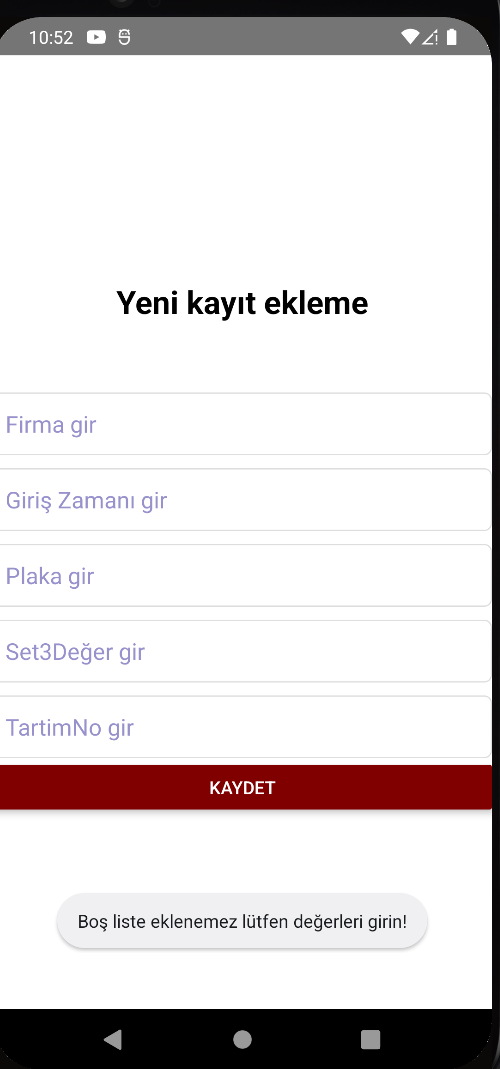


Figure 2.3 The warning message that comes when we do not make any changes and press the record button on the new record screen.

When adding a new record, the texts were showing white and not visible, I made them black. It also gives a warning if we save it without making any changes.

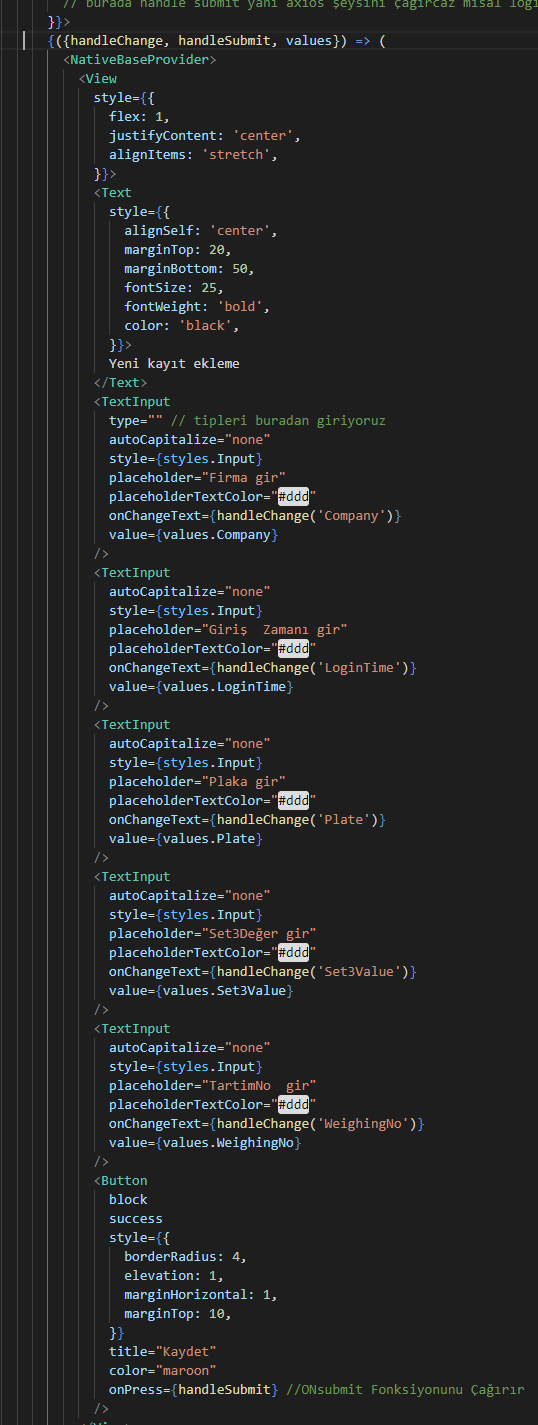
****

Figure 2.4 NewRecordModifyPopUp rest of the code.

**4.3 Delete vehicle from the list**

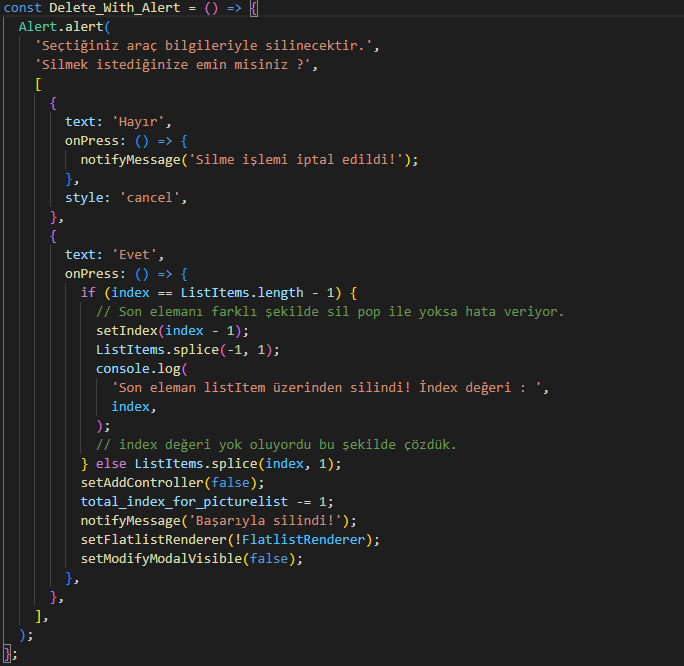


Figure 3.1 The code responsible for the deletion process in the edit screen.

Using the splice method with arrays in javascript, we deleted the vehicle in the selected index with the index prop returned from the flatlist.



Figure 3.2 Pop-up screen after pressing the delete button on the edit screen.

As we can see in the photo above, it asks us for confirmation to delete the vehicle. If we press yes, it deletes the vehicle from the list. If we press no, it does not make any changes, and the warning message "deletion has been canceled" appears on the screen.

**4.4 Modify Vehicle values with Pop-Up Modal**

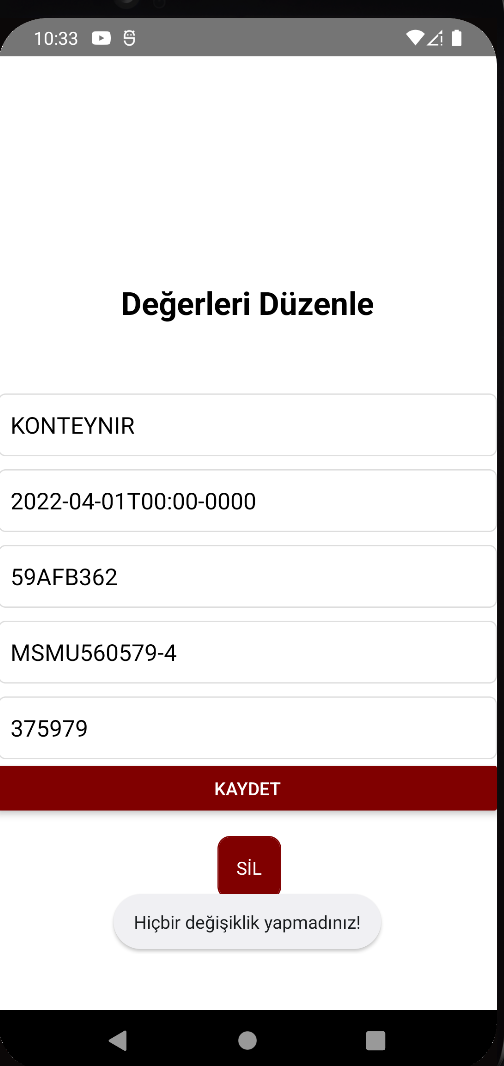


Figure 4.1 The warning message that comes when we do not make any changes and press the record button on the edit screen.

If we press save without making any changes, it gives a warning as follows in the below photo. Also in this photo, we can see that the values appear on the screen, and then the delete button was added.

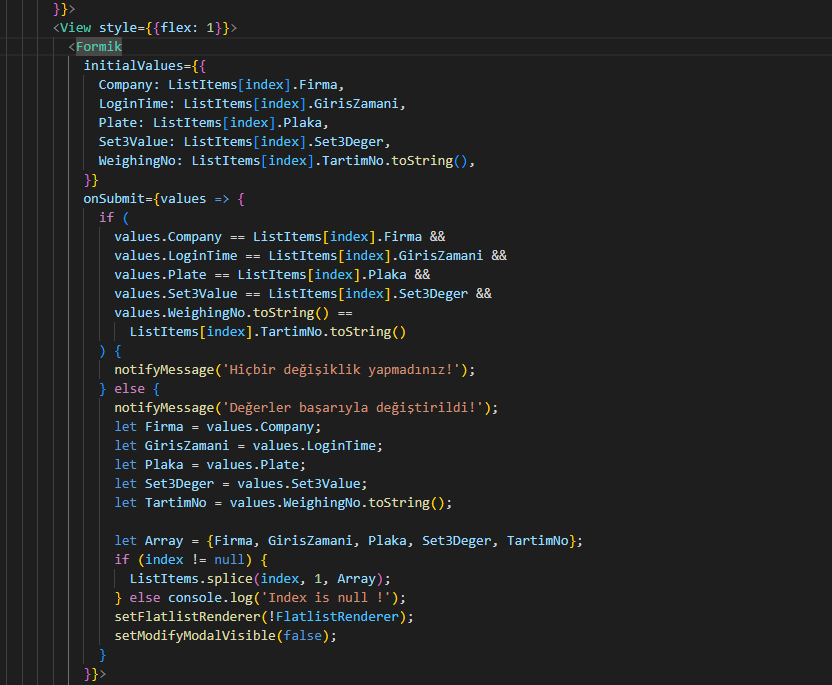


Figure 4.2 The code that makes the values appear on the edit screen at the beginning and shows the warning message if there are no changes after pressing the save button.

In the code above, we set the initial values as values of the vehicle instead of setting values as blank. And inside of the onsubmit function which is executed after pressing the save button, we checked in the if statement if we made changes or not. Everything else is the same as before.

**4.5 Add/Delete Details for each Vehicle**

We add a detail list for each vehicle inside of modify vehicle pop-up modal. We wanted to display the vehicles on that date according to the login time value in the vehicles.

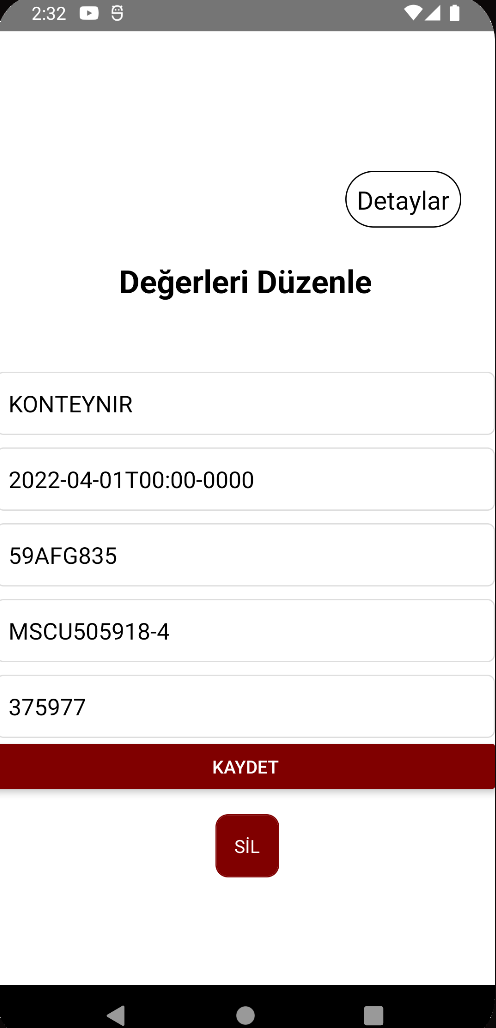


Figure 5.1 The button that is responsible for showing the details on the modify vehicle screen.

The details button is inside of the modify pop-up modal. When we press that details button which is called “detaylar” in there, it navigates us to the detail list screen, so we can add or delete details like modifying values we did before.

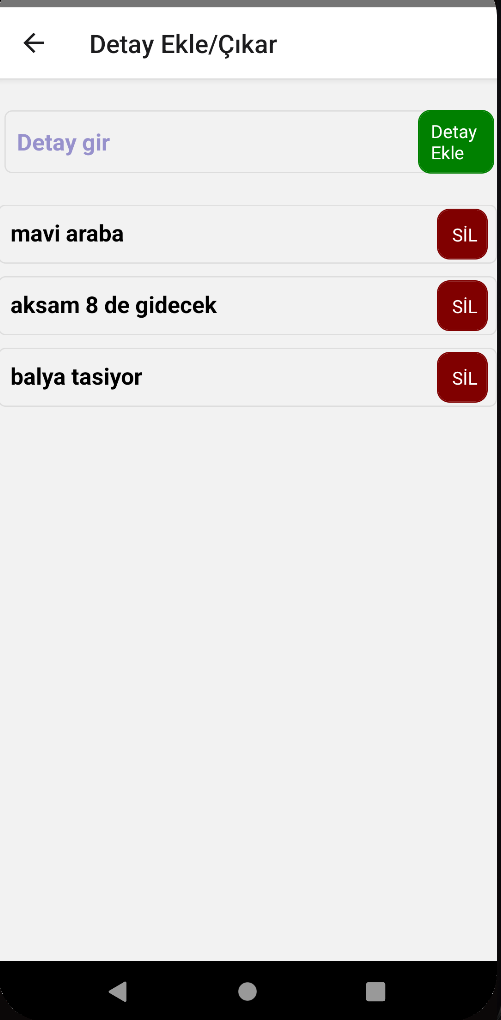


Figure 5.2 Add or remove the details screen that is responsible for details for each vehicle.

As we can see in the photo above, as we add details with the green add detail button, it is kept as a list, so it is displayed on the screen. We can add and remove details for each vehicle separately. The photo below shows us a confirmation message for deleting details from the detail list that we created before.

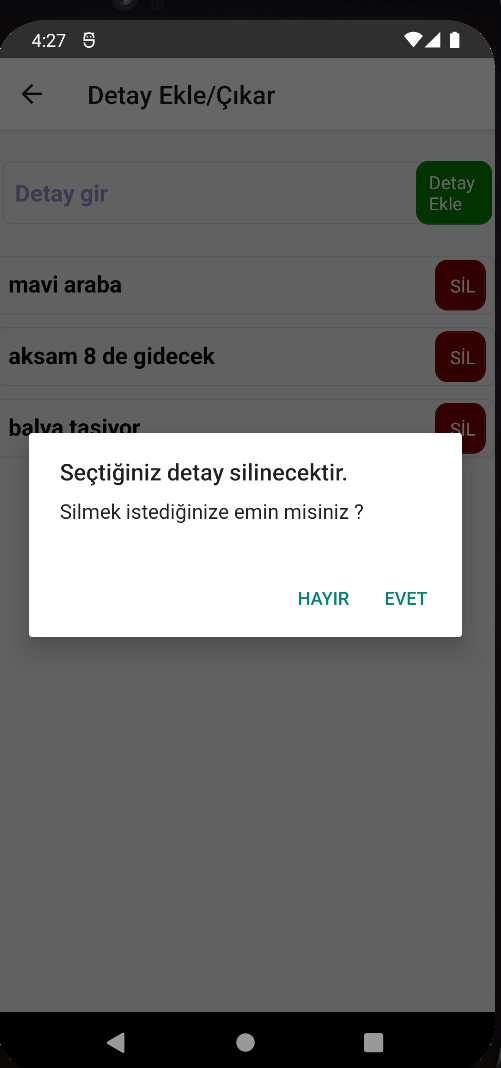


Figure 5.3 Confirmation message that comes after pressing the delete button.

When we press the yes button, the selected detail is deleted both from the list and from the screen. Now let's explain and examine the code parts of these.

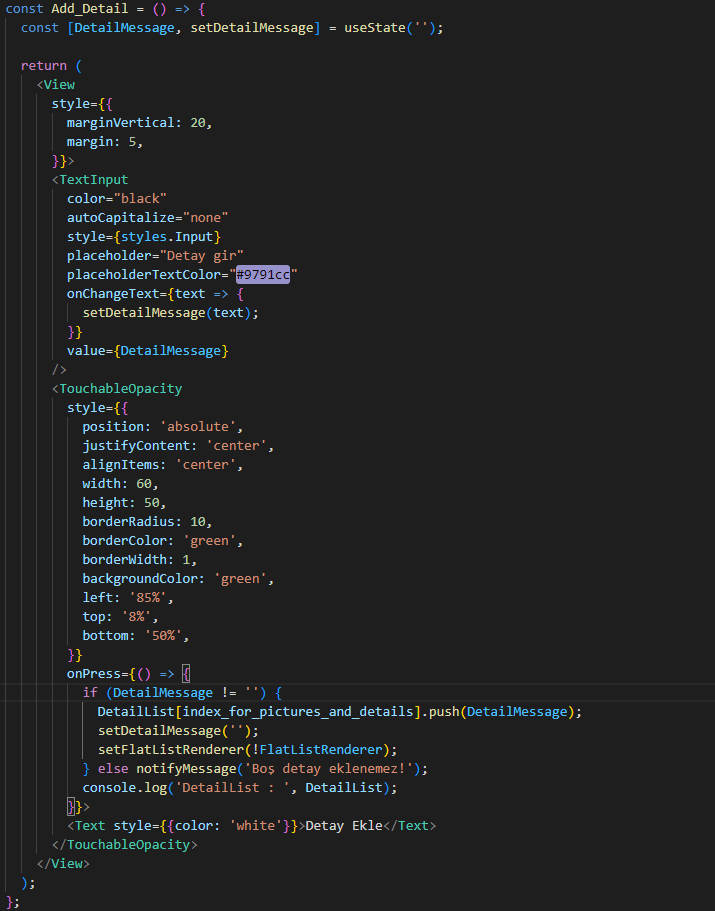


Figure 5.4 The code that is responsible for the green add detail button.

In the code above first, we created the DetailMessage variable that holds what we see on the screen, the message with the useState hook. The TextInput component is the field in which we write the detail message. TouchableOpacity is the green button that is called “add detail”. And inside of the button, we have an onpress prop that is responsible for what happened after pressing that button. First, it checks whether the detail message is null or not. If it is null it displays a warning message as “empty detail can not be added”.

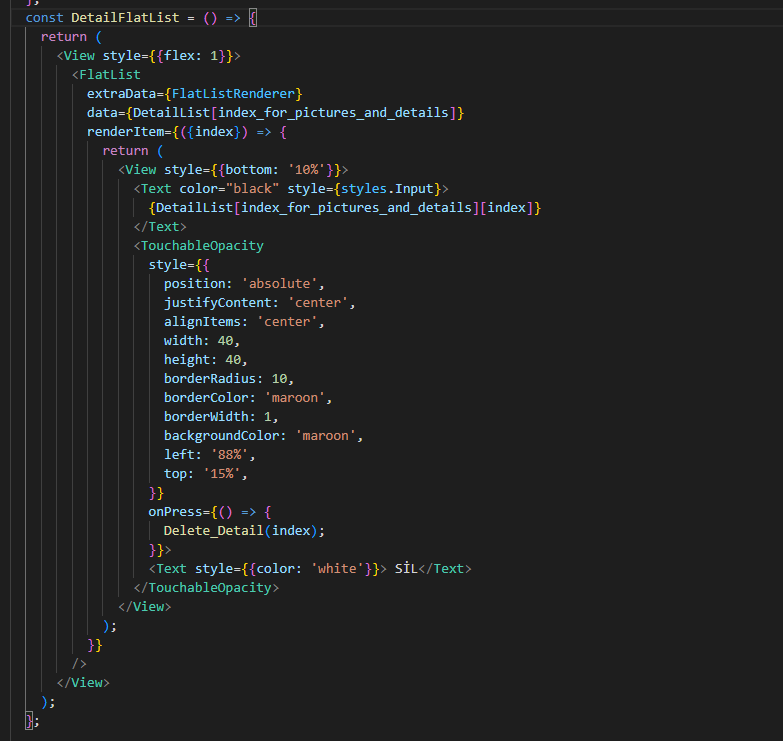


Figure 5.5 The code that is responsible for the List of the detail messages that are printed on the screen.

This is a classic FlatList component in react-native. Data prop is the DetailList array that we used to store all detail messages. And “index for pictures and details” is the variable that holds the index for the selected vehicle so we can add or delete that vehicle’s array. RenderItem is the prop that is responsible for how all details are shown on the screen. It is related to the style part. We have a “Sil” button that calls the Delete\_Detail function when we press it.



Figure 5.6 Delete Detail function that is responsible for when we press delete.

In the code above first, we have an alert that asks us if we are sure. If we are sure, which is yes, it deletes that detail message with index from array thanks to the splice method in javascript. Then it calls flatlistrenderer for immediate change on the screen. And the bottom part of the code is the main return code that is shown on the screen.

**4.6 DatePicker for displaying vehicles according to the selected date**

We create a date variable that is responsible for setting the date. We have the DateTimePicker method which is responsible for displaying the date picker on the top of the screen. . By default, the current time is set when we open the app. We have an open variable that is responsible for the visibility of the date picker. In short, it lets us display time on the screen.



Figure 6.1 The code that is responsible for the date selection filter at the top of the screen.

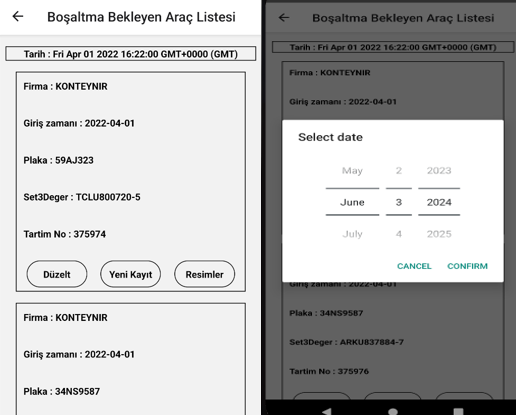


Figure 6.2 Date picker that shows the date on the screen when we press.

We can choose the date we want and the vehicles on that date are displayed on the screen. If there are no vehicles on that date it shows us a “no vehicle found in that date please select another” alert message The date we choose relates to the login time variable in the vehicles.

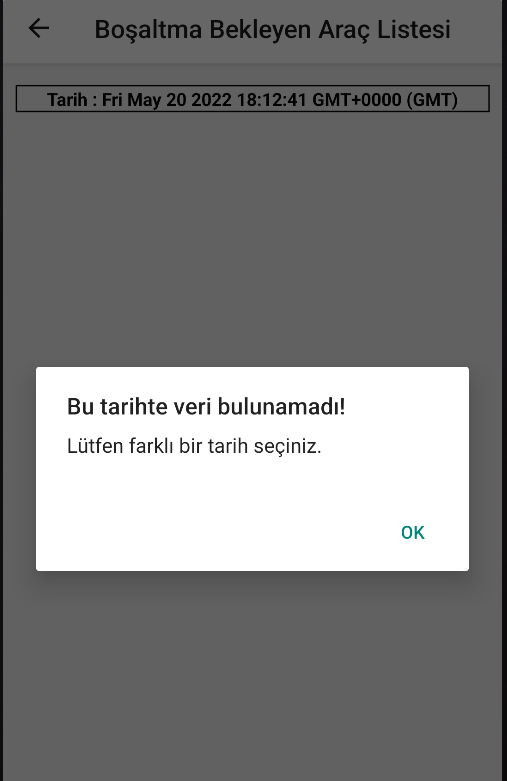


Figure 6.3 No vehicle found on selected date pop-up message.

Now I continue to explain the code parts of these.

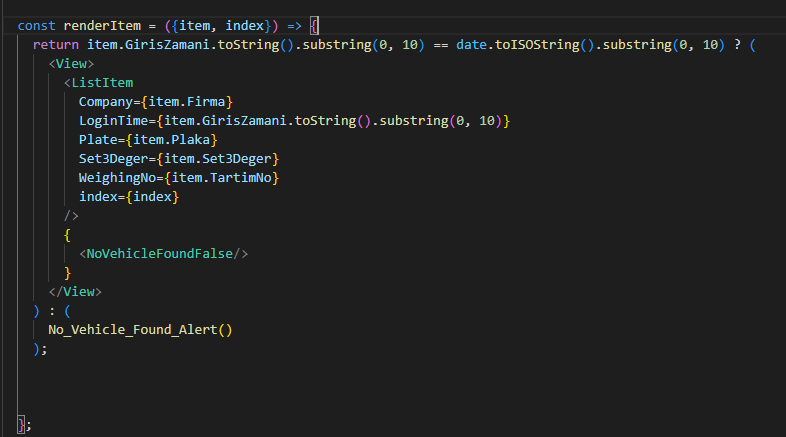


Figure 6.4 The code that decides whether the vehicles will appear on the screen according to their login times.

RenderItem is the prop that is responsible for how each vehicle will be displayed, as I said earlier. I use the ternary operator for deciding whether it will be displayed or not according to its date. I convert that Login time variable to the string and I take the first 10 characters as a substring from them. Because I don't wanna include hour minute and second parts. I also convert the date variable to the ISO date format to match the login time format. So if the vehicle’s login time value is equal to the date that we selected it prints that vehicle as a ListItem. Otherwise, if there are no vehicles related to that time, it prints a Novehiclefound alert on the screen.

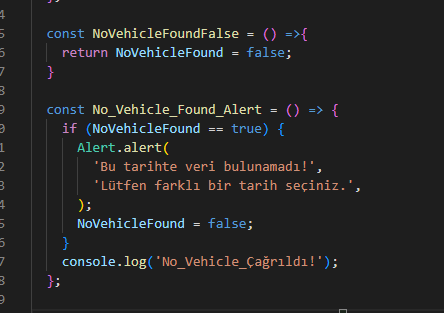


Figure 6.5 NoVehicleFound alert that appears on the screen when there are no vehicles found at that selected date.

**5. ADD PICTURE**

When we press the “Resimler” Button on Vehicle Waiting List it navigates us to the Pictures Screen. This is the screen in the below image. We have 2 buttons one of them is responsible for taking the photo from the camera and the other one is responsible for choosing from the gallery.

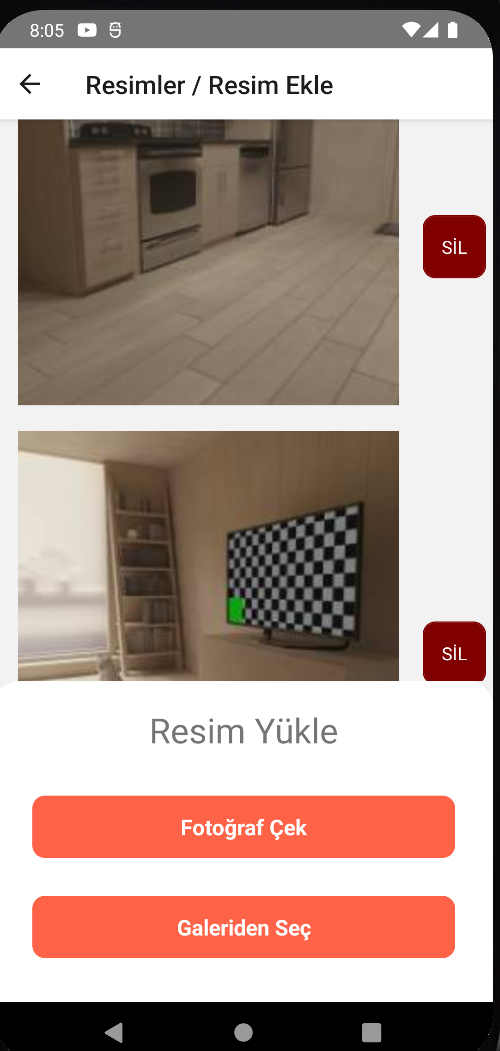


Figure 1.1 Pictures Screen Layout.

**5.1 Take Photo With Camera**

When we press to take a photo first it asks permission to access the camera. If we allow that it opens like that below. After taking a photo it allows us to edit the photo by rotating, cropping or adjusting the aspect ratio, etc. The image is cropped by default. We set the width and height as 300 pixels to fit the screen.

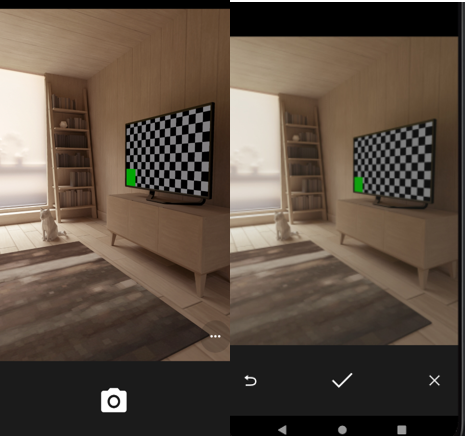
****

Figure 2.1 Take Photo With Camera UI.

****

Figure 2.2 Edit Photo as crop, rotate, aspect ratio in Camera UI.

Let’s examine the code part. We have 2 parts, First one is responsible for displaying selected or taken with a camera to show these images with ImageListBox Component. The bottom part of the screen is created by combining the Take Photo and Choose From Gallery methods into one component in **Figure 2.4**

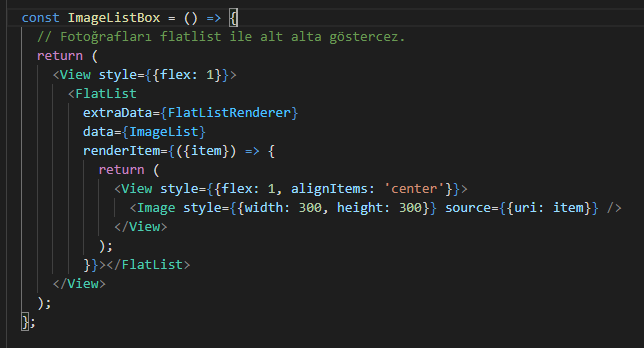


Figure 2.3 FlatList for displaying all images that we select or take a photo of.

We have an empty ImageList array. When we take a photo or select from the gallery we push that image.path into the array and display all the photos with FlatList.

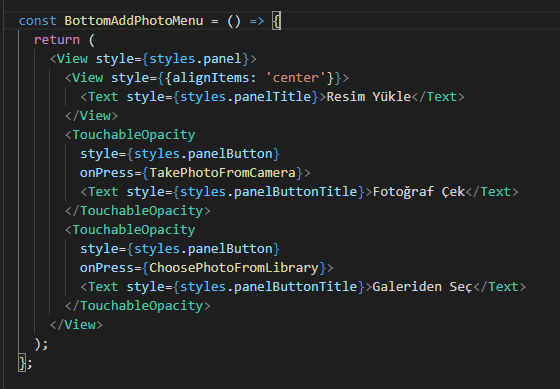


Figure 2.4 Bottom part of the Screen code that is responsible for taking a photo or choosing from gallery.

BottomAddPhotoMenu has 2 Buttons and each one of them calls a different method that is written in yellow. Now we talk about the takephotoFromCamera Part. I add if the condition to check the image is null or not to control if the user decides to give up on taking the photo or not.

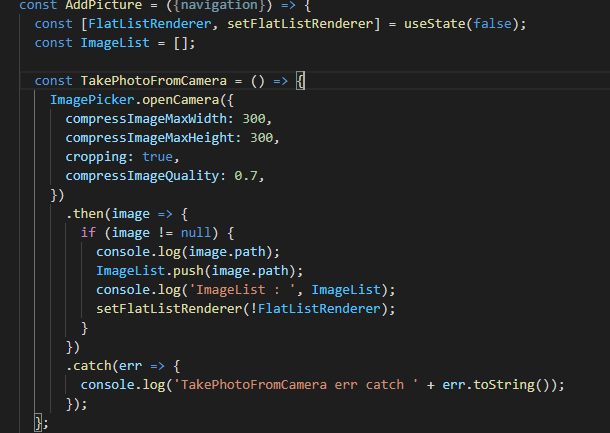


Figure 2.5 TakePhotoFromCamera method code.

We use the ImagePicker library in react native because it also crops for us automatically. We set the cropping values depending on us. With .then it returns the image object. This object has some properties like an image.path image.size etc. we only take an image.path . After taking a photo we push that path into the array in the below image as an output. We call flatlistrenderer for making changes immediately.



Figure 2.6 After Taking a Photo the console output.

**5.2 Choose Image From Gallery**

We have a choose photo from the gallery method as I mentioned earlier. And it works the same as the taking photo method. We can set a multiline prop for selecting multiple images but I don’t prefer that. I add if the condition to check the image is null or not to control if the user decides to give up on selecting a photo or not.

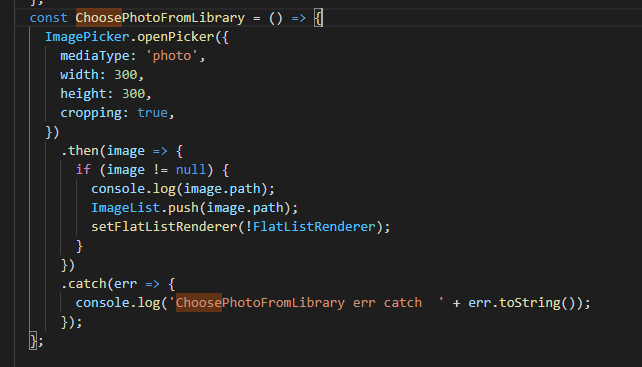


Figure 3.1 ChoosePhotoFromGallery code.

We set the width height and add this image to the array again. After Selecting a photo it goes to the edit screen again then it is added to the array until we press apply button.

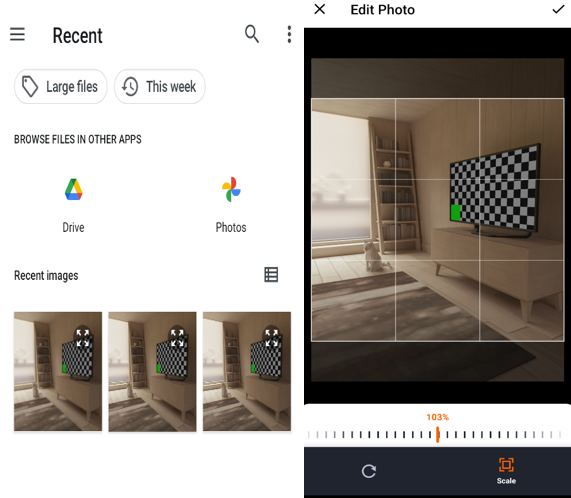
****

Figure 3.2 ChoosePhotoFromGallery in application UI.

**5.3 Delete Picture**

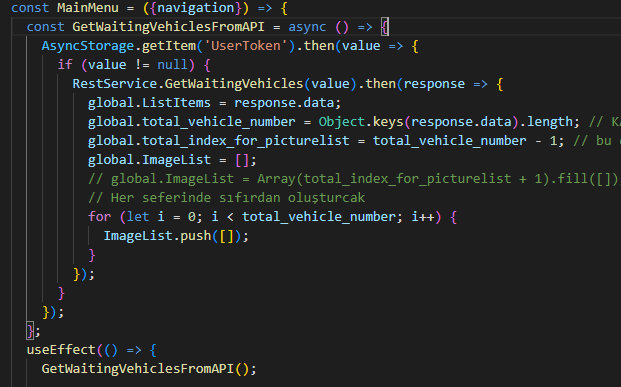


Figure 4.1 Updated code that creates vehicle and image list on the Main Menu screen.

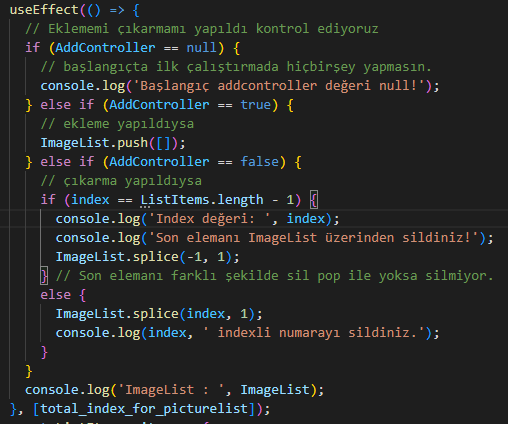


Figure 4.2 The UseEffect hook, which is triggered by the change of the total\_index\_for\_picture\_list value after the addition or subtraction operations, and controls whether the addition or subtraction is done with add controller state.

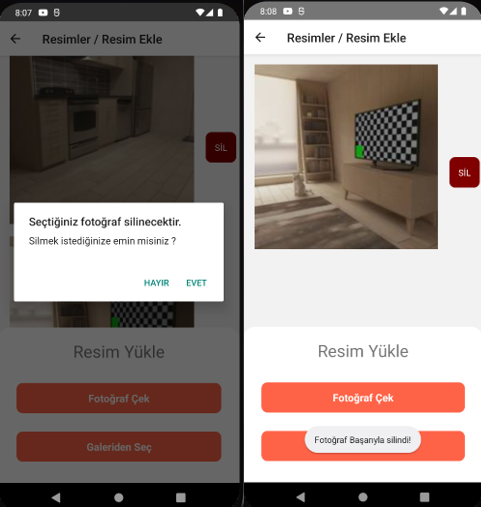


Figure 4.3 Confirmation message before deleting the image and the appearance after deleting the image.

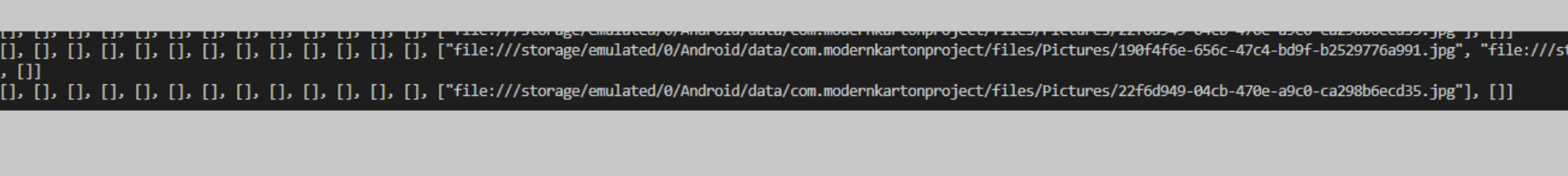


Figure 4.4 ImageList is shown in the console after adding and removing images.

As we see on these images we can successfully add or remove images from the ImageList which is synchronized with the Vehicle List array. Now I explain the code parts of these.

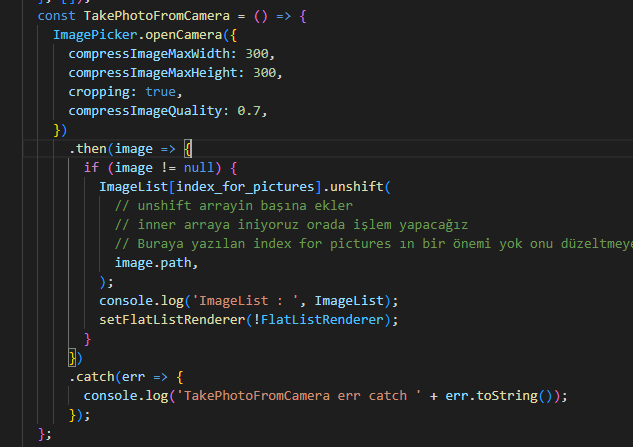


Figure 4.5 Updated TakePhotoFromCamera method which adds the image to the beginning of the ImageList array with unshift method.

ChoosePhotoFromLibrary method also works the same as the code above. We take index\_for\_pictures from the flatlist as a global variable. So we can access that variable from everywhere.

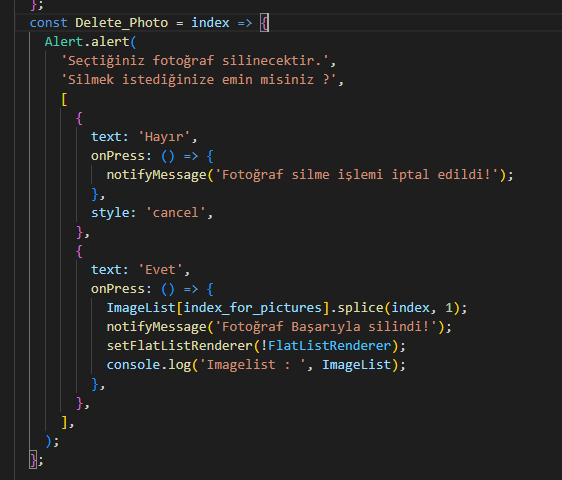


Figure 4.6 Delete photo method for deleting photos from ImageList.

This code is responsible for image deletion from vehicles. When we press the “SİL” button this method is called and executed with the specific array that returns from the flatlist. We use the classic splice method from javascript to delete that photo. And we call flatlistrenderer for extradata prop that provides re-render the list immediately.

**6. MobX for State-Management**

I found a bug in my app. When we entered the values in the new record or modify screen, it lost focus and did not allow us to type input. Therefore I asked my advisor. And he told me that I didn’t use the state management module in my project. I either use MobX or Redux for state management. They suggest me use MobX since it is easy to use and requires less effort and also my Project is not large enough for redux. I made big changes after realizing that critical bug. And for the solution, first I integrated MobX into my Project. As we see in the next photowe use the makeObservable method for defining observables and actions for MobX. Observable defines a trackable field that stores the state. The action marks a method as an action that will modify the state. We define our ListItems array which is responsible for holding Vehicle Data as an observable. We define our modify, add, and delete vehicle operations as an action since they directly manipulate ListItem array data. So instead of accessing it as a global variable, now we can access it from the DataStore class. Also instead of using a pop-up modal for manipulating vehicle data, now we navigate to the New Record Screen or Modify Screen. I also change my UI element’s color. And after changing all of that, I realized the real-time performance improvements after testing my app on my phone. Now it is snappy and fast.



Figure 1.1 MobX integration for state-management in React-Native.

**CONCLUSION**

At the beginning of this project, I did not know react-native and javascript. I learned to use node.js and the npm package that came with it. I also learned all of the technologies and methods that I mentioned earlier. In this project, I worked in multidisciplinary fields (logistics, transportation, software) and multidisciplinary team(back-end dev, front-end dev, tester). Every week I tested all apps and optimize my code. And every two weeks I fixed issues and bugs on my project. I got advice from my advisor when I can't solve a problem. I had front-end development experience with this project. I became familiar with Javascript and the design part of the application.