Bilkent University

Department of Computer Science

Senior Design Project

GrapeHealth

Low-Level Design Report

İlhami Özer, İbrahim Berker Kırdök, Murat Süerdem, Osman Sefa İbiş, Göktuğ Özdoğan
Supervisor: Prof. Dr. Halil Altay Güvenir
Jury Members: Prof. Dr. İbrahim Körpeoğlu - Doç. Dr. Özcan Öztürk
Innovation Expert: Kerem Erikçi (tarla.io)

Website: https://github.com/brker/GrapeHealth

Low-Level Design Report
Feb 18, 2018
This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Contents

1. Introduction

Grape is the most produced fruit in Turkey's market share. Turkey has an excessively link field for grape production and has great potential for viticulture. Also, Spain, France, Italy, and China are the world leaders of grape production. The importance of detecting grape diseases arises in thinking of the worldwide impact. The main purpose of our innovation is to be able to detect the five most common grape diseases to change the viticulture experience of the farmers to easily diagnose their grape products.

Farmers solely depend on agriculturalists to diagnose and cure their products, with our mobile application farmers can detect if their grape has a disease or not. The potential users are farmers, agricultural engineers, government's agricultural consultants, agricultural groups, grape merchants and drug companies or chemical stores. Our ambition is to help agricultural specialists and groups in being an additional resource to detect diseases. Grape merchants can decide whether or not the product they will buy has any of the common diseases. Farmers and drug companies can communicate more biologically to decide the accurate drug for the grapes. In some circumstances, chemical stores and farmers may not have the information to detect the diseases. Integrating the potential users to diagnose diseases will help grape production sector to communicate better. Detecting diseases will improve efficiency and quality by adding value to fresh and healthy grapes.

In this report, the low-level architecture of our system will be described in depth. The overview of the design and architecture of the system will be provided. First of all, after explaining our product in the introduction we will explain our design trade-offs. The packages of our system and the functionalities of each of them will be described along with class diagrams. The class interfaces in specified packages are provided with their descriptions of the functionalities they present.

Changes: An important change since the High-Level Design Report is, we decided to change our architecture into Redux. Since we are developing our application in React-Native, Redux architecture is a better suit than MVC architecture. Redux is an open-source JavaScript library for managing application state. It is most commonly used with libraries such as React or Angular for building user interfaces. Similar to (and inspired by) Facebook's Flux architecture [1]. Implementing a Redux model will be a bigger challenge for us, however, it will be good practice for all of us.

1.1 Object Design Trade-offs

1.1.1 Extensibility vs. Compatibility

GrapeHealth application focuses only on the grape plant, so for the future of this application we consider to extend this application by adding other plants into our disease detecting concern. Therefore, as a design approach we value extensibility because the application has the potential to extend in the market share and the application can be adapted to many other plants with ease. On the other hand, compatibility is important for the application to reach every possible user. Implementing with reach native allows us to release our application in both platforms (IOS & Android) in one code base fashion. For close future we value compatibility over extensibility because it will affect the early stages which are more important. For the future approach extensibility will become more important as a design tradeoff.

1.1.2 Complexity vs. Performance

The application requires user inputs (grape photo and chatbot answers) to produce a query and both of the input types must be handled in a fast manner to make the application perform to enhance user experience. During implementation, instead of using complex algorithms we will focus on response time and best matching as a performance criteria to maximize our performance. By establishing this design approach, the waiting time and user experience will be greater.

1.1.3 Functionality vs. Usability

GrapeHealth will be used by agricultural society who are related with grapes therefore considering the target users, we have to consider different ages adaptation with our application to make it user friendly. The UI of our application and the ease of using our application should accomplish these aspects. GrapeHealth will also be a functional application. Functionalities such as depicting graphical statistics to the queries the user inputs, warning about the possible threats nearby the user's grape field and learn about treatments for the grape related diseases. Considering functionality and usability aspects, we will value these design approaches the same and the application will be implemented considering this balance.

1.1.4 Security vs. Cost

Reusability of our implementation is an important aspect for the future design approach of our product since the product is extendible the later development becomes important. Reusability of our implementation will consider adding other plants and adapting the system to the new plants. However, for the early stages of our initial product minimizing the cost becomes more important but we will also consider the reusability of our code for future improvements.

1.2 Interface Documentation Guidelines

Throughout this report, we used the interface documentation guidelines in Figure 1 for each class. Thanks to this design, classes will be more understandable by the reader. As the figure below shows, the first row is the name of the Class and if it extends any other class or not. The second row is a description of what this class does. Later on, all properties and then all methods of the class is shown at the interface. The figure below is the outline.

Class Name / extends Class
Description of the class
Properties
All properties of the class
Methods
All methods of the class

Figure 1: Interface For Packages

1.3 Engineering Standards

IEEE citation guidelines are followed to specify references in reports. Unified Modelling Language (UML) design principles is used for visualizing system decomposition and underlying structure of the system through diagrams, use cases, subsystem decomposition and scenarios. Both IEEE guidelines and UML design principles are used to make reports more understandable.

1.4 Definitions, Acronyms, and Abbreviations

API: Application Programming Interface
FTP: File Transfer Protocol
HTTP: Hypertext Transfer Protocol
MVC: Model-View-Controller
TCP: Transmission Control Protocol
UI: User Interface
UML: Unified Modeling Language
SQL: Structured Query Language

2. Packages

Our program has 4 packages in total. View, Action and Reducer are in the Client side of the program while Learning Package is in Server side.

2.1 Client Packages

For the Client side of the application, we will use Redux architecture. In order to do that we construct the application three parts: Views, Actions and Reducers. Figure 2 below shows how the architecture works. When view part of a page takes an action, this action is sent to the action file of the page. This part is done what needs to be done (ie. *loginWithEmail()* function takes email and password and call firebase function to determine whether information are valid.) Then the reducer part compares previous and the changed state and if any change is found state is changed and sent to the view.

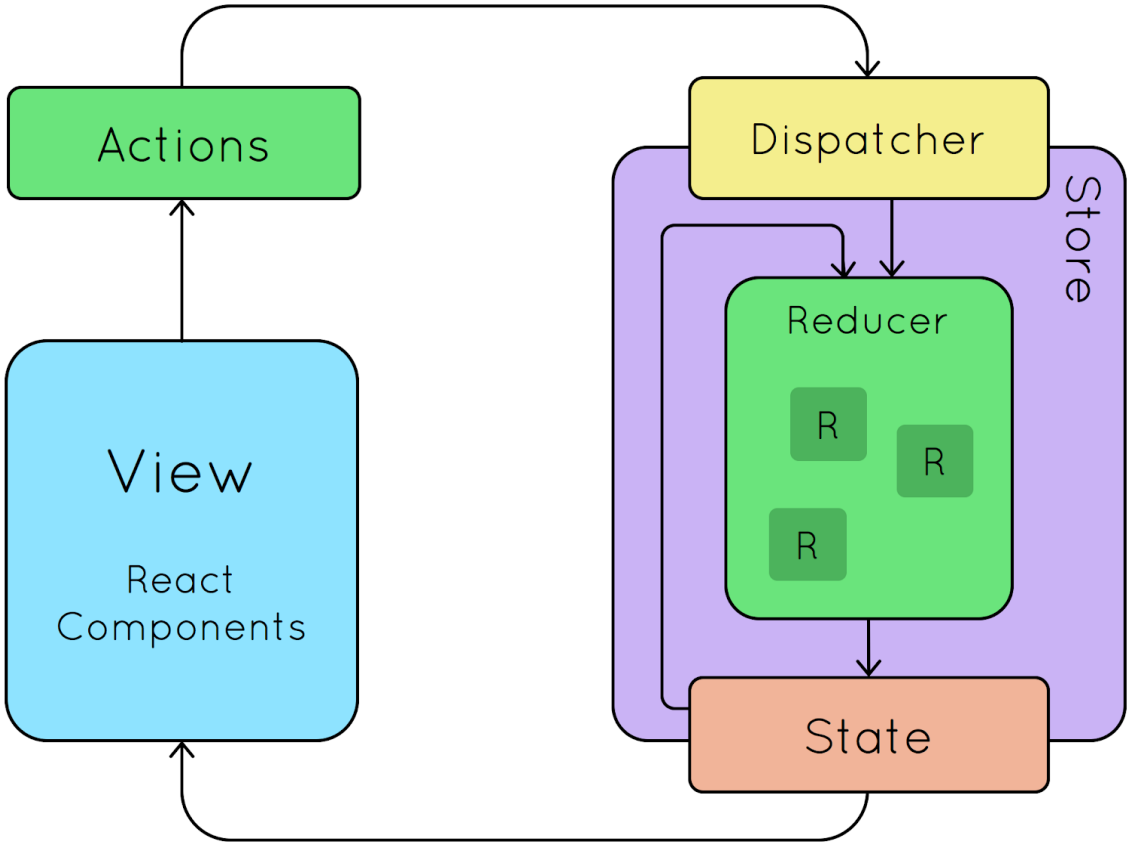


Figure 2: Redux Architecture [2]

2.1.1 View Package

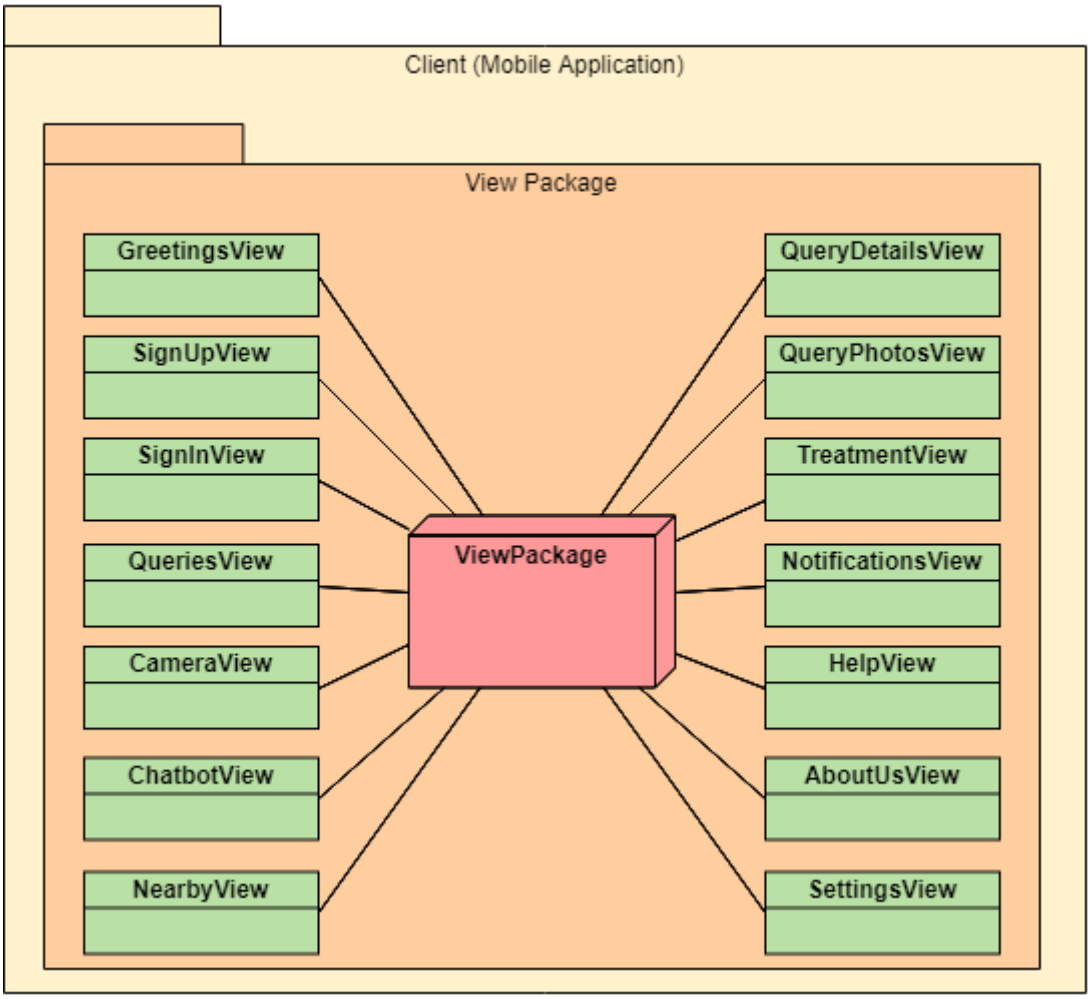
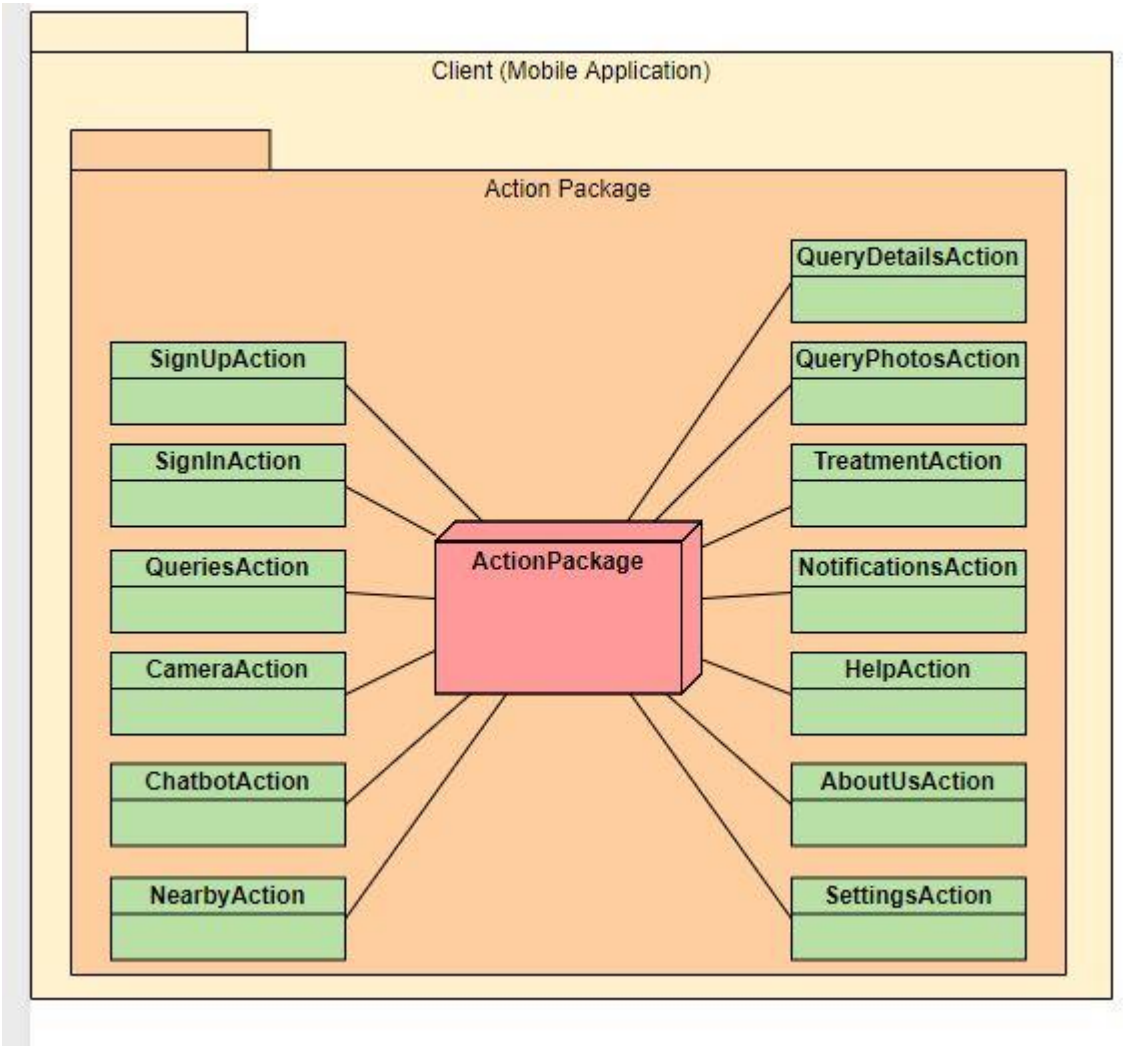


Figure 3: View Package

View package is responsible for user-interface of the Client side. All these files extend Component from React library. For each View, we have an Action and a Reducer class as well. View Package also contains some components like NavigationBar, TopPanel, Button, Card, Header etc. These components also extend Component from React library. They are either fix panels (NavigationBar, TopPanel) or templates (Button, Card) for View classes to shape them. We did not show these components in the diagram above, because they are not in Redux architecture, which means they do not have Action-Reducer classes for them. Besides, these type of components are very generic and we have lots of them.

2.1.2 Action Package



Action Package send data from application and database to reducer. Also action package changes the state of reducer package. With parameters, action package methods search databases and retrieve related data for user or store information of users in database system. When changes are detected in TextInput it changes the state of reducer.

2.1.3 Reducer Package

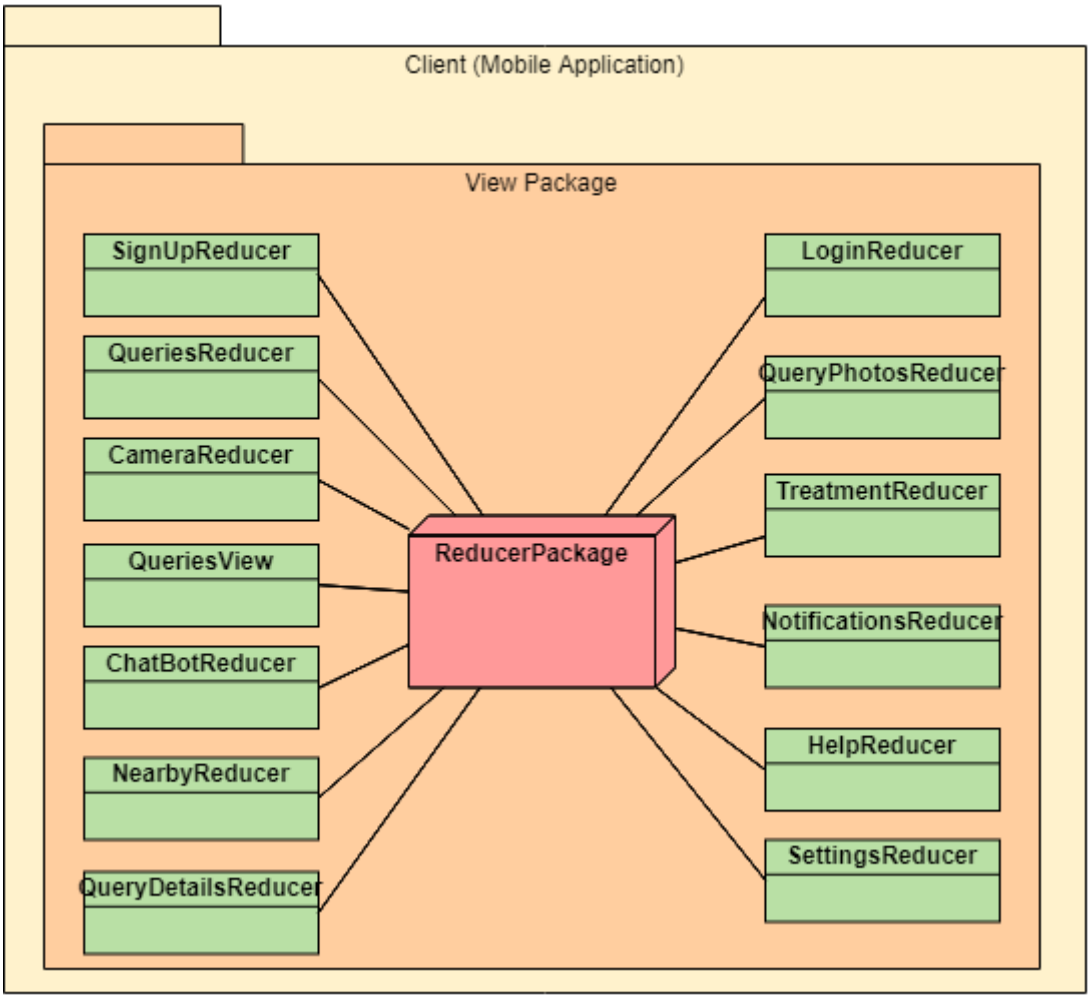
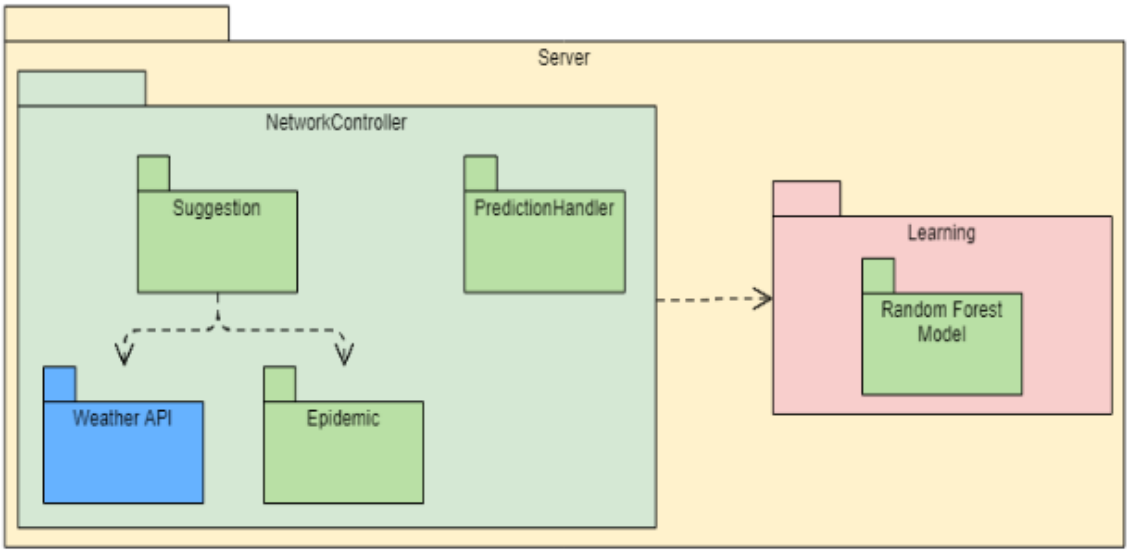


Figure 4: ReducerPackage

This package contains all file which contains states. When an action is triggered and dispatched, reducer works and determine which action is triggered. Then if there should be any change on the state according to this action, it makes it. As an example, when *loginWithEmail()* function is triggered, login succeeded and dispatched, *LoginReducer* determines that this action is triggered and changes LOGIN_STATE ‘s user property.

2.2 Server



2.2.1 Learning Package

The random forest model is trained in the learning package in the server side. On the mobile phones, GrapeHealth will take the photo from users and send it to the server, and it analyze data and make prediction.

2.2.2 NetworkController Package

Network Controller manages query results through PredictionHandler which handle requests coming from client side and inform it back, and also location suggestions for clients. Suggestions are based on location of the user. It notifies user about epidemics or weather. When an epidemic occurred in an field, it notifies all users close to there. We are planning to use OpenWeatherMap API to get current weather and forecasts [8]. It shows weather conditions to user while suggesting it is good or bad time to spraying. Windy or rainy days are not suitable for spraying.

3. Class Interfaces
3.1 Client

3.1.1 Greetings

class GreetingsView extends Component
View part of the application. When user enter the application greetings view will be displayed.
Properties
-
Methods

render() //render the view and its components
const mapStateToProps= ({greetingsViewReducer }) //changes state at the store

Figure 5: GreetingsView Class Interface of GrapeHealth

3.1.2 Sign Up

class SignUp View extends Component
View part of the signUp. The boxes which users type their information will be displayed and information will be passed to the signUp action.
Properties
Methods

render() //render the view and its components
const mapStateToProps= ({SignUPViewReducer }) //changes state at the store

Figure 6: SignUpView View Class Interface of GrapeHealth

class SignUp Action
Action part of signUp that takes information from users and send them to databases to write to databases and change the state of signUp reducer.
Properties
Methods

signUp({name,surname,email,password})
// take information from users and save them in database
// and send information of users to server as query
nameChanged( ) //trace name change in TextInput
surnameChanged() // trace surname change in TextInput
emailChanged() // trace email change in TextInput
passwordChanged() // trace password change in TextInput
reTypePasswordChanged() // trace password change in TextInput
signUpSuccess() //change user state
signUpFail() //show error message to user

Figure 7: SignUpAction Class Interface of GrapeHealth

class SignUpReducer
Reducer part of signUp that has state(includes information of users) and a method to determine which action is triggered.
Properties
INFORMATION_SIGNUP: {name,surname, email,password,password retype} //a JSON to store Information of users
Methods
default function (state = USER_INFORMATION, action)

Figure 8: SignUpReducer Class Interface of GrapeHealth

3.1.3 Sign In

class LoginView extends Component
View part of Login shows the login screen.
Properties
clickLoginWithEmail({email,password}) clickLoginWithGoogle() clickLoginWithFacebook()
Methods

render() //render the view and its components
const mapStateToProps = ({ loginReducer }) //changes state at the store

Figure 9: LoginView Class Interface of GrapeHealth

class LoginAction
Action part of Login, sign in the user to the system via Google, Facebook or Email.
Properties
Methods

emailChange() //trace email change in TexInput
passwordChange() //trace password change in TexInput
loginWithEmail({ email, password }) //login user with email and password
loginWithGoogle() //login user via Google Account
loginWithFacebook() //login user via Facebook Account
loginSuccess() //change the state of the user
loginFail() //shows alert message and error to user

Figure 10: LoginAction Class Interface of GrapeHealth

class LoginReducer
Reducer part of Login that has state(includes login information and user) and a method to determine which action is trigerred.
Properties
LOGIN_STATE: {} //a JSON to login information and user
Methods
default function (state = LOGIN_STATE, action)

Figure 11: LoginReducer Class Interface of GrapeHealth

3.1.4 Queries

class QueriesView extends Component
View part of the Queries. Old Queries of users display with this view.
Properties
Methods

render() //render the view and its components
const mapStateToProps= ({QueriesReducer }) //changes state at the store

Figure 12: QueriesView Class Interface of GrapeHealth

class QueriesAction
Action part of Queries which initially takes user which is session id and display all queries made from that user.
Properties
Methods

getQueries({userId}) //get Queries made from specific user
listQueries({queiryID}) //list all queries made from specific ***user***

Figure 13: QueriesActionClass Interface of GrapeHealth

class signUpReducer
Reducer part of signUp that has state(includes questions answers) and a method to determine which action is triggered.
Properties
QUERIES: {} // a JSON to store details of query
Methods
default function (state = QUERIES, action)

Figure 14: QueriesReducer Class Interface of GrapeHealth

3.1.5 Camera

class CameraView extends Component
View part of Camera that shows the camera of the device to take three ptotos to send for a query. It sends photos to CameraAction to change state by CameraReducer.
Properties
Methods

render() //render the view and its components
const mapStateToProps = ({ cameraReducer }) //changes state at the store

Figure 15: CameraView Class Interface of GrapeHealth

class CameraAction
Action part of Camera that takes photos and send them as query.
Properties
Methods

takePhoto() //change state of CameraReducer each time a photo is taken
sendQuery() //send all three photos as query

Figure 16: CameraAction Class Interface of GrapeHealth

class CameraReducer
Reducer part of Camera that has state(includes photos) and a method to determine which action is trigerred.
Properties
PHOTOS: {} //a JSON to store photos
Methods
default function (state = PHOTOS, action)

Figure 17: CameraReducer Class Interface of GrapeHealth

3.1.6 Chatbot

class ChatBotView extends Component
View part of ChatBot shows questions and get answers to send ChatBotAction to change answers state in ChatBotReducer.
Properties
Methods

render() //render the view and its components
const mapStateToProps = ({ chatBotReducer}) //changes state at the store

Figure 18: ChatBotView Class Interface of GrapeHealth

class ChatBotAction
Action part of ChatBot that takes questions from database and send answers to ChatBotReducer to change the state.
Properties
Methods

getQuestions() //change state of ChatBotReducer each time qquestion is answered
saveAnswers() //save answers in ChatBotReducer
sendAnswers() //send all answers to server as query

Figure 19: ChatBotAction Class Interface of GrapeHealth

class ChatBotReducer
Reducer part of ChatBot that has state(includes questions answers) and a method to determine which action is trigerred.
Properties
QUESTIONS_ANSWERS: {} //a JSON to store Questions and Answers
Methods
default function (state = QUESTIONS_ANSWERS, action)

Figure 20: ChatBotReducer Class Interface of GrapeHealth

3.1.7 Nearby

class Nearby extends Component
View part of Nearby shows a map and on it nearby diseases or warnings.
Properties
Methods

render() //render the view and its components
const mapStateToProps = ({ NearbyReducer}) //changes state at the store

Figure 21: NearbyView Class Interface of GrapeHealth

class NearbyAction
Action part of Nearby that takes nearby notification points and its details from database.
Properties
Methods

getNearby() //all points on the map
getDetails() //get details of a point when its is touched

Figure 22: NearbyAction Class Interface of GrapeHealth

class NearbyReducer
Reducer part of Nearby that has state(includes nearby points) and a method to determine which action is trigerred.
Properties
NEARBY_POINTS: {} //a JSON to store Nearby Points
Methods
default function (state = NEARBY_POINTS, action)

Figure 23: NearbyReducer Class Interface of GrapeHealth

3.1.8 Query Details

class QueryDetailsView extends Component
View part of *QueryDetails* which shows details about the selected query.
Properties
-
Methods

render() //render the view and its components
const mapStateToProps = ({ queryDetailsReducer }) //changes state at the store

Figure 24: QueryDetailsView Class Interface of GrapeHealth

class QueryDetailsAction
Action part of *QueryDetails* which initially takes a query id from *Queries* and display it in details.
Properties
-
Methods
getQuery(id) //gets all information about selected query to display

Figure 25: QueryDetailsAction Class Interface of GrapeHealth

class QueryDetailsReducer
Reducer part of *QueryDetails* which has a state(includes details) and a method to determine which action is triggered.
Properties
DETAILS: {} // a JSON to store details of query
Methods
default function (state = DETAILS, action)

Figure 26: QueryDetailsReducer Class Interface of GrapeHealth

3.1.9 Query Photos

class QueryPhotosView extends Component
View part of *QueryPhotos* which shows photos of the selected query.
Properties
-
Methods

render() //render the view and its components
const mapStateToProps = ({ queryDetailsReducer }) //changes state at the store

Figure 27: QueryPhotosView Class Interface of GrapeHealth

class QueryPhotosAction
Action part of *QueryPhotos* which initially takes a query id from *QueryDetails* and display photos of that query.
Properties
-
Methods
getQueryPhotos(id) //gets all photos of selected query to display

Figure 28: QueryPhotosAction Class Interface of GrapeHealth

class QueryPhotosReducer
Reducer part of *QueryPhotos* which has a state(includes photos) and a method to determine which action is triggered.
Properties
PHOTOS: {} //a JSON to store photos
Methods
default function (state = PHOTOS, action)

Figure 29: QueryPhotosReducer Class Interface of GrapeHealth

3.1.10 Treatment

class TreatmentView extends Component
View part of *Treatment* which shows photos of the selected query.
Properties
-
Methods

render() //render the view and its components
const mapStateToProps = ({ queryDetailsReducer }) //changes state at the store

Figure 30: TreatmentView Class Interface of GrapeHealth

class QueryPhotosAction
Action part of *Treatment* which initially takes a query id from *QueryDetails* and display suggested treatment if any.
Properties
-
Methods

getQueryTreatment(id) //gets suggested treatment of selected query to display
suggestedApplyDate() //shows suggested apply date to user

Figure 31: TreatmentAction Class Interface of GrapeHealth

class QueryPhotosReducer
Reducer part of *Treatment* which has a state(includes treatment) and a method to determine which action is triggered.
Properties
TREATMENT: {} //a JSON to store treatment
Methods
default function (state = TREATMENT, action)

Figure 32: TreatmentReducer Class Interface of GrapeHealth

3.1.11 Notifications

class NotificationView extends Component
View part of Notification shows all notification around user about and disease or weather forecast that could affect the user.
Properties
Methods

render() //render the view and its components
const mapStateToProps = ({ notificationReducer }) //changes state at the store

Figure 33: NotificationView Class Interface of GrapeHealth

class NotificationAction
Action part of Notification that takes all notifications and weather forecast and inform the user.
Properties
Methods
getNotifications() //get all notifications around the user

Figure 34: NotificationAction Class Interface of GrapeHealth

class NotificationReducer
Reducer part of Notification that has state(includes notification details) and a method to determine which action is trigerred.
Properties
NOTIFICATIONS: {} //a JSON to store Notifications
Methods
default function (state = NOTIFICATIONS, action)

Figure 35: NotificationReducer Class Interface of GrapeHealth

3.1.12 Help

class HelpView extends Component
View part of Help shows a guideline how a user could use the application.
Properties
Methods

render() //render the view and its components
const mapStateToProps = ({ helpReducer }) //changes state at the store

Figure 36: HelpView Class Interface of GrapeHealth

class HelpAction
Action part of Help that fetches all guideline from database.
Properties
Methods
getHelp() //get guideline from database

Figure 37: HelpAction Class Interface of GrapeHealth

class HelpReducer
Reducer part of Help that has state(includes guideline) and a method to determine which action is trigerred.
Properties
GUIDELINE: {} //a JSON to store all parts of guideline
Methods
default function (state = GUIDELINE, action)

Figure 38: HelpReducer Class Interface of GrapeHealth

3.1.13 About Us

class AboutUsView extends Component
View part of AboutUs shows information about developer team.
Properties
Methods
render() //render the view and its components

Figure 39: AboutUsView Class Interface of GrapeHealth

3.1.14 Settings

class SettingsView extends Component
View part of Settings gives user to change his/her profile information, email and password.
Properties
Methods

render() //render the view and its components
const mapStateToProps = ({ settingsReducer }) //changes state at the store

Figure 40: SettingsView Class Interface of GrapeHealth

class SettingsAction
Action part of Setting update user information.
Properties
Methods

nameChange() //trace name change in TexInput
surnameChange() //trace surname change in TexInput
emailChange() //trace email change in TexInput
passwordChange() //trace password change in TexInput
saveChanges() //update changes and change state

Figure 41: SettingsAction Class Interface of GrapeHealth

class SettingsReducer
Reducer part of Settings that has state(includes profile information) and a method to determine which action is trigerred.
Properties
SETTINGS_STATE: {} //a JSON to store profile information
Methods
default function (state = SETTINGS_STATE, action)

Figure 42: SettingsReducer Class Interface of GrapeHealth

3.2 Server

3.2.1 NetworkController

class PredictionHandler
It take input data(photo) and give it to the Learning Model in order to take prediction which represents query result.
Properties
prediction: # disease of the leaf
Methods
trainNewModel(): getPrediction():

Figure 43: PredictionHandler Class Interface of GrapeHealth

3.2.2 Learning

class Learning
Model is trained in this class and provide prediction for given data.
Properties
X_train: # train data features y_train: # train data labels X_test: # test data features y_test: # test data labels randomForestModel: # model to fit randomForestModel_prediction: # predictions for test data randomForestModel_accuracy: # accuracy of the model n_estimators: # for tuning hyperparameters Max_features: # for tuning hyperparameters Max_depth: # for tuning hyperparameters disease: # label of the given data
Methods
train(): predict():

Figure 42: Learning Class Interface of GrapeHealth

4. Glossary

Client	The part of the system the users interact with.
Server	The part of the system responsible for logical operations, scheduling, and data management.
Redux	Redux is an open-source JavaScript library for managing application state. It is most commonly used with libraries such as React or Angular for building user interfaces. Similar to (and inspired by) Facebook's Flux architecture**.**
React Native	React Native is a JavaScript framework for building native mobile apps. It uses the React framework and offers a large number of inbuilt components and APIs.
Keychain	Keychain is the password management system in macOS, developed by Apple.
PlantNet	PlantNet is an image sharing and retrieval application for the identification of plants.

5. References

1. 1. Redux (JavaScript library). (2019, January 27). Retrieved from https://en.wikipedia.org/wiki/Redux_(JavaScript_library)
2. 2. React Redux: Building Modern Web Apps with the ArcGIS JS API. (n.d.). Retrieved from https://www.esri.com/arcgis-blog/products/js-api-arcgis/3d-gis/react-redux-building-modern-web-apps-with-the-arcgis-js-api/?rmedium=redirect&rsource=blogs.esri.com/esri/arcgis/2017/09/08/arcgis-js-api-4-x-with-reactredux
3. "iOS Distribution and iOS Market Share," *Apteligent | Mobile Application Performance and User Experience*. [Online]. Available: https://data.apteligent.com/ios/. [Accessed: 15-Oct-2018].
4. "Distribution dashboard | Android Developers," *Android Developers*. [Online]. Available: https://developer.android.com/about/dashboards/. [Accessed: 15-Oct-2018].
5. K. Erikçi, "Interview with Co-founder of ErikTronik," 10-Oct-2018.
6. S. Srivatsa, *A Summary of Network Traffic Monitoring and Analysis Techniques*. [Online]. Available: https://www.cse.wustl.edu/~jain/cse571-14/ftp/android_security/index.html#sec2.1. [Accessed: 30-Dec-2018].
7. OpenWeatherMap.org, "Weather API," *openweathermap*. [Online]. Available: https://openweathermap.org/api. [Accessed: 25-Dec-2018].