



Politecnico di Torino

Ingegneria Informatica(Computer Engineering)

Tesi di Laurea Magistrale

**Development of a Cross-Platform Mobile
Application and an Image Cloud Storage for an
IoT System**

Advisors

Prof. Daniele Trincherò

Ing. Giovanni Paolo Colucci

Ana Laura TULA

APRIL 2018

Summary

The iXemWine platform, developed by the research group from the iXem Laboratory, uses wireless technology applied to precision agriculture in rural areas that allows wine producers to monitor their vineyards through a web application.

Given the widespread use of smartphones, there was a need to create a mobile application to help the visualization the data collected from different sensors in a meaningful way. The two main mobile operating systems, Android and iOS, uses both distinct languages to create native applications which are fast and optimized, but this leads to a duplication of the code base. There are cross-platform solutions aim to simplify this problem by using a single programming language for developing applications that are targeted for multiple platforms.

The main goal of this project is to develop a cross-platform mobile application with a focus on the Android operating system using React Native, a JavaScript framework for writing natively rendering mobile applications for iOS and Android.

The goal of the second part of the project is to investigate the different options for cloud storage service and develop a possible solution to store the pictures taken by the sensor's camera and visualize them in the application.

Acknowledgment

I would like to thank my family, for their unconditional support, for guiding me, for their infinite patience and for allowing me to find my path in life. This thesis is the culmination of a long journey that has certainly not been easy one, and would not have been possible without them.

To my mother, for being my role model and pushing me to be better and what a better way to thank you that writing this thesis in English.

To my father, who has shaped the person I am today and who took me to my first computer lesson as a kid without knowing I would follow that course.

To my siblings, for putting up with me and always having my back.

To my grandmother, mi nona, who is always with me.

To my aunt, mi tía Ester, Juana and mi late grandmother, mi nona Clara, who were a big part of my life growing up and always showed their support.

To my lifelong friends who, despite the time and distance, are always present and to the new ones, my family during these years, with whom I shared this experience and have unforgettable memories.

At last but not least, I would like to thank the whole iXem Laboratory team, specially my thesis Advisors, Prof. Trincherro and Ing. Giovanni Colucci, for their guidance throughout the development of the thesis.

Table of Contents

1	Introduction.....	1
1.1	Introduction.....	1
1.2	Thesis Goals.....	1
1.3	General overview of the iXemWine Platform.....	1
2	Cross-platform vs Native.....	3
2.1	Native.....	3
	Benefits.....	3
	Drawbacks.....	3
2.2	Cross-platform.....	3
	Drawbacks.....	4
	Benefits.....	4
3	Javascript.....	5
3.1	ECMAScript.....	5
3.2	Transpilers.....	5
3.3	Polyfill.....	6
3.4	Modules.....	6
3.5	Bundling.....	7
3.6	Minification.....	7
3.7	JavaScript environments.....	7
	3.7.1 Node.....	8
3.8	Metro.....	9
	3.8.1 The Journey to a Bundle.....	9
4	React.....	11
4.1	The DOM.....	11
4.2	React.....	11
	4.2.1 Component.....	12
	4.2.2 Component Lifecycle.....	13
	Mount.....	14
	Update.....	14
	Unmount.....	14
	4.2.3 Refs.....	14
	Creating Refs.....	15
	Accessing Refs.....	15
	4.2.4 Higher-Order Components.....	16
	4.2.5 Reconciliation.....	16
5	React Native.....	18
5.1	Internals.....	18
	5.1.1 Threading Model.....	18
	5.1.2 Bridge.....	19
	5.1.3 JavaScript Environment.....	19
	5.1.4 JavaScript Syntax Transformers and Pollyfills.....	19
5.2	Differences with React Web.....	20

5.3	Components and APIs.....	20
5.3.1	Basic Components.....	20
5.3.2	User Interface.....	22
5.3.3	List Views.....	22
5.3.3.1	FlatList.....	22
5.3.3.2	SectionList.....	23
5.3.4	iOS/Android specific.....	23
5.3.5	Others.....	24
5.3.5.1	Animated.....	24
	Interpolation.....	25
	Handling gestures and other events.....	25
5.3.6	Style.....	25
5.3.7	Direct Manipulation.....	26
5.3.8	Installation.....	26
5.3.8.1	Dependencies.....	26
5.3.9	React Native CLI Commands.....	28
5.3.10	The CLI.....	30
5.3.11	Debugging.....	30
5.3.12	Linking libraries.....	30
6	Data Visualization.....	32
6.1	Scalable Vector Graphics (SVG).....	32
6.2	D3.js.....	33
6.2.1	D3-scales.....	33
6.2.2	D3-shape.....	33
6.2.3	D3-array.....	33
7	Implementation.....	34
7.1	Actors of the system.....	34
7.1.1	Community.....	35
7.2	Functionality.....	35
7.2.1	Authenticate.....	36
7.2.2	Visualize the list of Vineyards.....	37
7.2.3	Visualize the Nodes and Cameras inside a Vineyard.....	38
7.2.4	Visualize the Sensors data in Charts.....	39
7.2.5	Visualize the pictures taken by a Camera.....	42
7.2.6	Search for vineyards in the Community.....	43
7.2.7	Edit preferences ans see the profile.....	44
8	Application Navigation.....	46
8.1	React Navigation.....	46
8.1.1	Navigator.....	46
8.1.2	Authentication flow.....	46
8.1.3	App containers.....	47
8.1.4	Navigation Prop.....	47
8.1.5	Passing parameters to routes.....	48
8.1.6	Handling state.....	48
8.1.7	Navigation lifecycle.....	48
8.1.8	Application navigation.....	49

9	Application State.....	51
9.1	Flux.....	51
9.2	Redux.....	52
9.2.1	Combining Reducers.....	53
9.2.2	React-redux.....	54
9.2.3	Async Requests.....	54
9.2.4	Redux Thunk.....	54
9.2.5	Persisting State.....	55
9.2.6	Application State.....	55
10	Web Services.....	58
10.1.1	Promises:.....	58
10.1.2	Async/Await.....	58
10.1.3	JSON Web Token.....	59
10.1.4	Moment.....	60
10.1.5	Data Transformations.....	61
11	Deploy.....	62
11.1	Generating Signed APK.....	62
11.2	Generate the release APK.....	62
12	Image Cloud Storage.....	63
12.1	Own cloud infrastructure(In-house-servers).....	63
12.2	Amazon cloud service.....	63
12.3	Functionalities to provide.....	64
12.3.1	Authentication.....	65
12.4	Flask.....	66
Routing.....		66
HTTP methods.....		66
The Response object.....		67
12.5	SQLAlchemy.....	68
12.6	Amazon Simple Storage Service.....	68
12.7	AWS Command Line Interface.....	69
12.8	Boto3.....	69
Creating Clients.....		69
12.8.1.1	Upload a File.....	70
12.8.1.2	Generate a pre-signed URL.....	70
12.9	Proposed Solution.....	71
13	Conclusion.....	72
13.1	Future work.....	72
14	References.....	73

Index of Illustrations

Illustration 1.3.1: Overview of the Application.....	2
Illustration 4.2.1: Lifecycle diagram.....	13
Illustration 4.2.2: Commonly used lifecycle methods.....	13
Illustration 4.2.3: High Order Component.....	16
Illustration 5.1.1: The Threads and the Bridge.....	18
Illustration 5.3.1: ScrollView parts.....	21
Illustration 5.3.2: Android platforms installed.....	27
Illustration 5.3.3: Project structure.....	28
Illustration 5.3.4: Options for commands available in the current Project.....	30
Illustration 7.1.1: Representation of a user's data.....	35
Illustration 7.2.1: Use Cases.....	36
Illustration 7.2.2: LoginScreen: Authentication of the user.....	37
Illustration 7.2.3: Visualization of the list of vineyards.....	38
Illustration 7.2.4: Visualization of sensors and cameras in a vineyard.....	39
Illustration 7.2.5: Visualization of the temperature and the dew point of a Temperature-Humidity Sensor.....	41
Illustration 7.2.6: Visualization of the humidity of a Temperature-Humidity Sensor	41
Illustration 7.2.7: Visualization of a Two Layer Leaf Wetness Sensor.....	42
Illustration 7.2.8: Photo grid visualization of the pictures taken by a camera.....	43
Illustration 7.2.9: Search and visualization of the vineyard's position.....	44
Illustration 7.2.10: User's profile and language change screen.....	45
Illustration 8.1.1: Application Navigation.....	50
Illustration 9.1.1: Flux achitecture.....	52
Illustration 9.2.1: Redux flow.....	53
Illustration 9.2.2: Application State.....	56
Illustration 9.2.3: Actions and Screens.....	57
Illustration 10.1: Token-based Authentication.....	60
Illustration 12.2.1: Camera backend architecture.....	64
Illustration 12.5.1: Database for Cloud Image System.....	68
Illustration 12.9.1: Proposed Solution for Cloud Image System.....	71

Index of Tables

Table 4.2.1: Types of React Components.....	12
Table 5.3.1: Main properties of Flex.....	26
Table 12.3.1: Backend functionalities.....	65

Index of Listings

Listing 4.2-1: Access a Ref.....	15
Listing 5.3-1: React Native CLI installation.....	28
Listing 5.3-2: Creation of a new React Native Project.....	28
Listing 5.3-3: Launch the application in an Android device or emulator.....	29
Listing 5.3-4: Launch the application in iOS device or emulator.....	29
Listing 5.3-5: Command to verify the connected smartphones.....	29
Listing 5.3-6: Linking a native library.....	31
Listing 8.1-1: Navigate to a another screen and pass parameters.....	48
Listing 8.1-2: Read a parameter.....	48
Listing 9.2-1: Function prototype for a middleware.....	54
Listing 10-1: Pseudo code for calculating the signature.....	59
Listing 10-2: Getting current date and time.....	60
Listing 10-3: Initialization with specific date.....	60
Listing 11.2-1: Commands to generate the APK.....	62
Listing 12.4-1: Minimal Flask application.....	66
Listing 12.4-2: Example of GET and POST method.....	67
Listing 12.4-3: Example of query string.....	67
Listing 12.4-4: make_response example.....	67
Listing 12.8-1: Create a client.....	69
Listing 12.8-2: Upload a file.....	70
Listing 12.8-3: Generate a presigned URL to return to the client.....	70

1 Introduction

1.1 Introduction

The Internet of Things (IoT)[1] provides efficient resource utilization while reducing the human intervention to increase the productivity. Within an IoT system a wireless sensor network (WSN)[2] can be used. A WSN is a network of devices that can communicate the information gathered from a monitored field through wireless links. The data is then forwarded through multiple nodes, and with a gateway, the data is at last forwarded to a Server, which exposes a RESTful API for the clients to use.

The aim of the iXemWine[3] platform is collecting data from the crops for precision agriculture. This allows wine producers to benefit from the advantages of IoT and to monitor their vineyards through a web application.

Given the widespread use of smartphones, there was a need to create a mobile application to help the visualization the data collected from different sensors in a meaningful way.

1.2 Thesis Goals

The main goal of this project is to develop a cross-platform mobile application with a focus on the Android operating system using React Native[4], a JavaScript[5] framework for writing natively rendering mobile applications for iOS[6] and Android[7].

The goal of the second part of the project is to investigate the different options for cloud storage service and develop a possible solution to store the pictures taken by the sensor's camera and visualize them in the application.

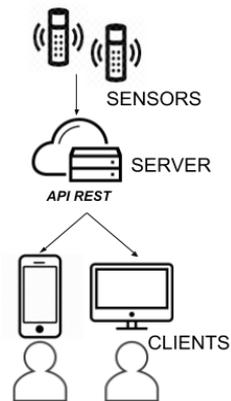
1.3 General overview of the iXemWine Platform

The platform iXemWine is a Low-Power Wide-Area Network[8], with sensors able to transmit information over long distances, even in non-line of sight, by means of unlicensed frequencies.

When data is received by Gateways, it is sequentially forwarded to the Network Server using standard TCP/IP[9] connections.

The application Server, written in Python[10], receives data by the Network Server using the MQTT[11] protocol. The server also provides a RESTful[12] API[13], as shown in Illustration 1.3.1, that can be used to retrieve all the information by means of HTTP[14] requests.

The REST API returns data in a language-independent format that clients can easily consume, this allows the interaction with many interfaces and a consistent user experience.



*Illustration
1.3.1: Overview
of the
Application*

2 Cross-platform vs Native

2.1 Native

Native applications are the ones that are developed exclusively for a specific platform and its Operating System in a language compatible with them, for example, the iOS devices use Objective-C[15] or Swift[16] while the Android devices uses Java[17] or Kotlin[18].

Benefits

- Developed for the specific platform, they are fast and responsive.
- The platform's software development kit (SDK) allows access to all the device APIs.
- Platform comes with familiar and original user interface (UI) components.
- Allows use of device-specific functionalities.
- Android and iOS have large communities behind them. If there's a frequent problem, it is highly likely to find a ready-made solution in the form of a third-party library or an API.

Drawbacks

- Requires separate development processes for every platform, since the code for one platform only works for that platform.
- With different code bases, it is always difficult to release same features on all platforms at the same time.
- The cost of development and maintenance of the mobile application is high.

2.2 Cross-platform

Cross-platform mobile applications are developed using an intermediate language that is not native to the device's operating system. This means that some, or all, of the code can be shared across platforms – for instance, across both iOS and Android. To build mobile applications, the cross-platform frameworks provides a platform-independent API that uses a programming language, like JavaScript that the developers use to build the mobile application, including the user interface, data persistence and business logic.

Those solutions can be split into two categories:

- HTML5 based (PhoneGap[19], Apache Cordova[20], Ionic[21]): Developers create mobile applications using web technology, they load a mobile browser in the application and perform all logic operations within that browser, offering also added functionalities that traditional web technology does not.
- Native Widget based (React Native): Developers can use JavaScript to construct their application from components that are subsequently mapped into the platform specific widgets. In order to use native widgets, the JavaScript part of the application has to communicate with the native part through a “bridge”.

Drawbacks

- Not as many third-party components, also some of them might behave differently than expected. A lot of components have small communities behind them, and the creators can't always update them frequently.
- Not as efficient as a native application.
- Limited device API access.

Benefits

- React Native provides nearly identical performance to native.
- React Native actually renders using its host platform's standard rendering APIs, which enables it to stand out from most existing methods of cross-platform application development.
- Currently popular among mobile application developers.
- Its ecosystem is expanding rapidly every day along with new features.
- The development effort is lower since there is a high percentage of the code that can be reused.

Taken into account all of this, a decision was reached to choose a cross-platform approach using the React Native framework for the development of the iXemWine mobile application.

3 Javascript

JavaScript is a high-level, interpreted programming language that conforms to the ECMAScript specification. It is characterized as dynamic, weakly typed, prototype-based and multi-paradigm. JavaScript is primarily a client-side language.

The vast majority of websites uses JavaScript, major web browsers have different dedicated JavaScript engines to interpret and execute the code (or uses some sort of lazy compilation), for example:

- Chrome[22] and Node[23]: V8[24]
- Firefox[25]: SpiderMonkey[26]
- Safari[27]: JavaScriptCore[28]
- Microsoft Edge[29]/IE[30]: Chakra[31]

They each implement the ECMAScript standard, but may differ for anything not defined by it.

3.1 ECMAScript

ECMAScript is the standard upon which JavaScript is based. ECMAScript 2015[32], also known as ES6, is a fundamental version of the standard. Since a long time passed between the previous release, this release is full of important new features and major changes in suggested best practices in developing JavaScript programs.

This update adds significant new syntax for writing complex applications, including classes and modules, which simplifies the defining of complex objects with their own prototypes. Other new features include iterators and for/of loops, Python-style generators, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, number and math enhancements, reflection, proxies (meta-programming for virtual objects and wrappers) and template literals.

3.2 Transpilers

Transpilers, or source-to-source compilers, are tools that read source code written in one programming language, and produce the equivalent code in another language.

As stated previously, every browser uses a different JavaScript engine and each one, has:

- different performance characteristics
- implements a different subset of ES2015 features
- is approaching full compliance with the spec at different rates.

That means that, some features works fine for those users running the most recent Chrome, Firefox, or Safari, but it won't work for users running older versions, or for anyone using Internet Explorer.

Kangax[33] created an ES6 compatibility table where developers can track the support of various ES6 features on various JavaScript engines, the ES6 compatibility table[34] shows that there was a clear progress, but it is not quite time to write ES6 directly. Instead, developers write source code in ES6, and let a transpiler translate it to plain ES5 that works in every browser.

One of the most popular tools is Babel[35], a tool-chain mainly used to convert ECMAScript 2015 code into a backwards compatible version of JavaScript that works both in current and older browsers or environments.

Babel also has plugins available to provide specific conversions used in web development. For example, developers working with React[36], can use Babel to convert JSX[37] (JavaScript eXtension) markup into JavaScript using the Babel preset "react".

3.3 Polyfill

When Babel compiles the code, is taking the syntax and running it through various syntax transforms in order to get browser compatible syntax. But it is not adding any new properties needed to the browser's global namespace or any JavaScript primitive. This can be achieved by using a polyfill.

A polyfill is code that defines a new object or method in browsers that don't support them natively. A polyfill can be used to implement various browser features other than ECMAScript standards, for example SVG[38], Canvas, Web Storage (local storage/session storage) among others.

3.4 Modules

Almost every language has a concept of modules — a way to include functionality declared in one file within another. Typically, a developer creates an encapsulated library of code responsible for handling related tasks. That library can be referenced by applications or other modules.

The benefits of using modules:

- code can be split into smaller files of self-contained functionality.
- The same modules can be shared across any number of applications.
- Code referencing a module understands it's a dependency. If the module file is changed or moved, the problem is immediately obvious.
- Module code helps eradicate naming conflicts.

ES6 introduced built-in modules that offers up a variety of alternatives for importing and exporting modules.

3.5 Bundling

Bundling is the process of following imported files and merging them into a single file: a “bundle”. In web development the bundling, or the “concatenation” of all the files into one big file (or a couple files as the case may be) is in order to reduce the number of requests. This is referred as the “build step” or “build process”.

3.6 Minification

Minification is the process of removing all unnecessary characters from the source codes of interpreted programming languages or markup languages without changing their functionality. These unnecessary characters usually include white space characters, new line characters, comments, and sometimes block delimiters, which are used to add readability to the code but are not required for it to execute. Minification reduces the size of the source code, making its transmission over a network more efficient.

3.7 JavaScript environments

Nowadays, there can be considered mainly three types of JavaScript environments:

- 1) Client-side browser JavaScript: Scripts written in JavaScript that are embedded in a web page's HTML and run client-side by a JavaScript engine in the user's web browser.
- 2) Client-side native JavaScript: Most devices can have a JavaScript run-time, therefore, developers can ship JavaScript files with Android/iOS/Desktop applications and then run them there. These engines also support adding

“hooks” from JavaScript into the native code, and that's how React Native provide its API.

- 3) Server-side JavaScript: Server-side JavaScript refers to JavaScript that runs on server-side and is therefore not downloaded to the browser. Server generally has two meanings:
- A piece of software that listens for network requests and then responds to them.
 - A computer running such a piece of software.

Node is a tool that can be either of those, and also allows to use JavaScript for non-server and non-web purposes.

3.7.1 Node

Node is a JavaScript run-time environment built on Chrome's V8 Javascript engine.

Generally, the server itself is run directly from Node (e.g. with the HTTP built-in module) rather than being embedded in another server like Apache[39] (as is most common for PHP[40]). In this case, a browser doesn't need to be involved at all. If one is, then it will probably be one acting as a client and making a request to the server.

A fourth case can be added to the previous environments:

- 4) Build-scripts running on runtimes like Node

Developers can use JavaScript to generate JavaScript files, that is, to build and combine several JavaScript files into a single file, and transpile it if needed. This process is how to bundle the files for (1) and (2) and maybe also (3).

Metro is a server-side build-script (which runs on Node) that is used by React Native to work as a:

- Development Server: Start a server that serves JavaScript code as a web page would (1 & 3). During the development stage it allows to iterate quickly on a device by connecting to the development server running on the machine.
- Bundler: Bundle all the JavaScript code in a native application that later can be installed on a mobile device (2).

3.8 Metro

Metro[41] is the development platform for React Native, it does that by exposing a HTTP server so clients, in this case, emulators or connected devices can communicate with it and it also exposes a Websocket[42] server so it can push updates into them.

In development, the bundle will come from the the development server. That way if the code is changed, the server will send a request to the client, through the Websocket, to download the new code or update the code on the fly. So, the bundle is dynamically generated from the source code.

In production, an offline bundle is used since the code is already on the device and does not need to be downloaded.

Metro is a JavaScript bundler. It takes in options, an entry file, and the output is a JavaScript file including all JavaScript files back.

Metro has three separate stages in its bundling process:

- **Resolution:** Metro needs to build a graph of all the modules that are required from the entry point. To find which file is required from another file, Metro uses a resolver. In reality this stage happens in parallel with the transformation stage.
- **Transformation:** All modules go through a transformer. A transformer is responsible for converting (transpiling) a module to a format that is understandable by the target platform (eg. React Native). Transformation of modules happens in parallel based on the amount of cores available.
- **Serialization:** As soon as all the modules have been transformed they will be serialized. A serializer combines the modules to generate one or multiple bundles. A bundle is literally a bundle of modules combined into a single JavaScript file.

3.8.1 The Journey to a Bundle

Metro is itself a package that is run on Node and that is the reason Node is a dependency of React Native. The main task is to build JavaScript code, and that also entails:

- **Monitoring all the files in the project:** Using the module `jest-haste-map`[43] from another project called `Jest`[44] gives the ability of monitoring the file system and emitting changes every time it detects changes. In order to achieve

this, it uses watchman[45] if it is installed. Watchman is an open source project from Facebook which monitor the file system as a daemon process so it reduces the start-up time. However if not installed, it falls back to fs.watch[46], this has a start-up overhead but after that is the same as using watchman.

- Transform the source code: Metro does what any other bundler do, which is to use Babel, the difference is the way it is executed. Transpilation is a very expensive process, most bundlers have a main process and execute the transpilation process one file after the other. Metro uses a different approach, the main process doesn't transform any file, but it spawns a set of child processes that are called workers. Files are sent in parallel to each of these workers, so the transpilation happens at the same time and then return the result back. Usually it spawns one worker per core.
- Store cache artifacts: Metro ships with an internal multi-layer cache located inside the Main process. If the file was never transpiled, it will transpile the file and save it in the cache.
- Build bundles: Metro produces bundles through serializers, which receive the graph and can manipulate them in any way.
- Execute code on devices.

4 React

4.1 The DOM

Browsers render HTML to a web page, before this happens they create a Document Object Model[47] of the page. The DOM is an object-oriented tree-like structure representation of the web page, constructed in memory, which can be modified with a scripting language such as JavaScript.

A HTML document can have encapsulated HTML content inside of other HTML content. The browser when loading the HTML document interrupts and parses this hierarchy to create a tree of node objects that simulates how the markup is encapsulated.

The problem with the DOM is that it is not optimized for dynamic applications. So, updating it slows the application when there are a several things to be changed; as the browser has to reapply all styles and render new HTML elements.

4.2 React

React is an open source JavaScript library created by Facebook for building user interfaces[48]. It introduces a declarative approach for building UIs that allows developers to:

- write declarative views that “react” to changes in data
- abstract complex problems into smaller modules called components

React is declarative, in the sense that allows developers to declare what they want and the library will take care of the DOM manipulation efficiently, without working directly with the browser API.

To minimize the number of costly DOM operations required to update the UI, React builds and maintains an internal representation of the rendered UI. It includes the React elements returned from the components. This representation lets React avoid creating DOM nodes and accessing existing ones beyond necessity, as that can be slower than operations on JavaScript objects. Sometimes it is referred to as a “virtual DOM”.

4.2.1 Component

Components are JavaScript functions or classes that returns a node (something React can render, e.g. a `<div/>`) and receives an object of the properties that are passed to the element.

These object, called props, are passed to a component and used to compute the returned node. Changes in these props will cause a re-computation of the returned node, which means a re-render. Unlike in HTML, these can be any JavaScript value.

During the development stage, React can validate the types of component props at runtime. This allows developers to ensure the correct passing of props and help document the component's APIs.

There are two main types of components in React: Class Components and Stateless Functional Components. The most obvious reason is that Class components are ES6 classes while Functional Components are functions. Additional differences are listed in the following Table 4.2.1:

Stateless Functional Component (SFC)	Class Component (React.Component)
<ul style="list-style-type: none"> • Simplest component: use when a state is not needed. • A function that takes props and returns a node. • Should be “pure” (it should not have any side effects like setting values, updating arrays, etc.) • Any change in props will cause the function to be re-invoked. 	<ul style="list-style-type: none"> • An abstract class that can be extended to behave the way developers need. • Have additional features that SFCs do not: <ul style="list-style-type: none"> ◦ Have instances ◦ Maintain their own state ◦ Have lifecycle methods (similar to hooks or event handlers) that are automatically invoked • Rendering becomes a function of props and class properties.

Table 4.2.1: Types of React Components

Class Components can have a state which adds an internally-managed storage to a component. State is a class property on the component instance, *this.state*, and can only be updated by invoking *this.setState()* and passing it an object to be merged, or a function of previous state. The calls to *setState()* calls are batched and run asynchronously.

Components are units containing both the rendering logic and the UI logic and instead of separating them, React puts markup in JavaScript. So, writing React means writing JSX, an XML-like syntax extension of JavaScript that transpiles to JavaScript, where lowercase tags are treated as HTML/SVG tags and uppercase are treated as custom components. Using JSX allows to write concise HTML/XML-like structures (e.g., DOM like tree structures) in the same file where is the JavaScript code, then Babel will transform these expressions into actual JavaScript code.

4.2.2 Component Lifecycle

The component lifecycle contains three major phases: mounting, updating and unmounting.

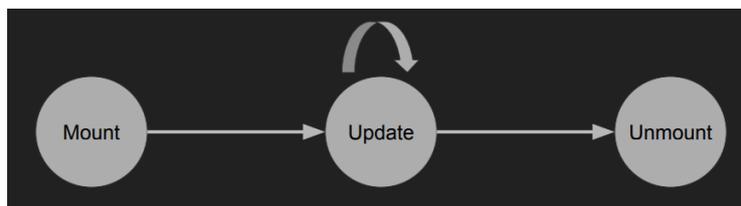


Illustration 4.2.1: Lifecycle diagram

The Illustration 4.2.2 [49] shows the order in which the most common lifecycle methods are called:

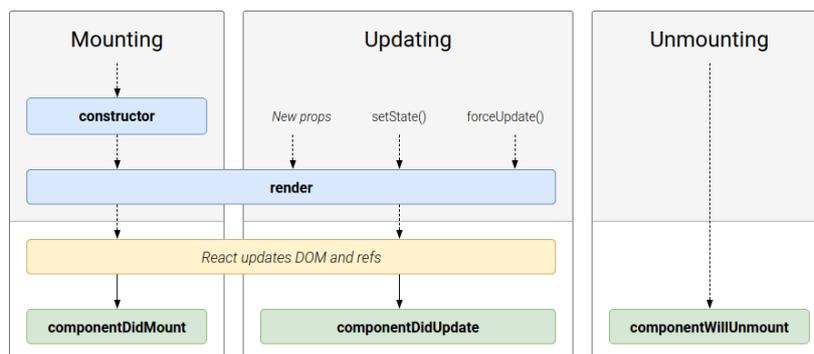


Illustration 4.2.2: Commonly used lifecycle methods

Mount

Since class-based components are classes, the first method that runs is the constructor method. Typically, is the place to initialize the component's state, or other class properties, bound methods, etc.

The `render()` method is the most used lifecycle method. It is in all React classes, this is because `render()` is the only required method within a class component. As the name suggests it handles the rendering of the component to the UI. It happens during the mounting and updating of the component.

React requires that the `render()` is pure. Pure functions are those that do not have any side-effects and will always return the same output when the same inputs are passed. This means that it is not possible to `setState()` within a `render()`.

Now that the component has been mounted and is ready, `componentDidMount()` is called. This is the place to initiate API calls, if data needs to be loaded from a remote endpoint. Unlike the `render()` method, `componentDidMount()` allows the use of `setState()`. Calling the `setState()` here will update state and cause another rendering but it will happen before the browser updates the UI. This is to ensure that the user will not see any UI updates with the double rendering.

Update

This phase is triggered every time the state or props changes, which cause a re-render. The method `componentDidUpdate()` can do anything that isn't needed for UI (network requests, etc.). In this lifecycle, `setState()` can be used, but it has to be wrapped it in a condition to check for state or prop changes from previous state. Incorrect usage of `setState()` can lead to an infinite loop.

Unmount

The unmounting phase is the last stage of the component lifecycle, so when a component is removed from the DOM, React invokes `componentWillUnmount()` right before it gets removed. This method is used to clean up any open connections such as WebSockets or intervals, remove event listeners. This component will never be re-rendered and because of that it doesn't make sense to call `setState()` during this lifecycle method.

4.2.3 Refs

Refs[50] provide a way to access DOM nodes or React elements created in the render method. In the typical React data flow, props are the only way that parent components interact with their children. To modify a child, re-render it with new

props. However, there are a few cases when modify a child outside of the typical dataflow, for example managing focus and text selection.

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. As Listing 4.2-1 shows, Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends
React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

Drawing 4.2.1: Create a Ref

Accessing Refs

When a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref:

```
const node = this.myRef.current;
```

Listing 4.2-1: Access a Ref

The value of the ref differs depending on the type of the node:

- When the ref attribute is used on an HTML element, the ref created in the constructor with `React.createRef()` receives the underlying DOM element as its current property.
- When the ref attribute is used on a custom class component, the ref object receives the mounted instance of the component as its current.

- Function components can not use the `ref` attribute because they don't have instances.

4.2.4 Higher-Order Components

A higher-order component[51] (HOC) is an advanced technique in React for reusing component logic. They are a pattern that emerges from React's compositional nature.

Concretely, a higher-order component is a function that takes a component and returns a new component. Since it returns a new component, it adds an extra layer of abstraction. In this layer can be added state, behavior, or even style, as shown in the Illustration 4.2.3.

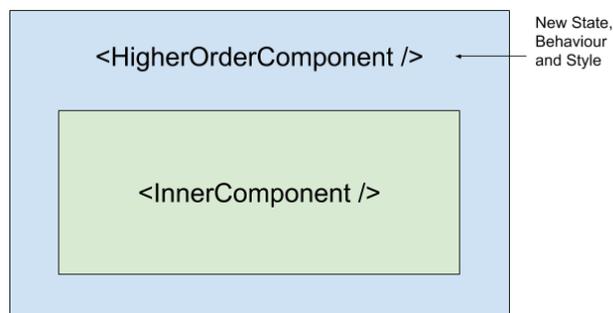


Illustration 4.2.3: High Order Component

Whereas a component transforms props into UI, a higher-order component transforms a component into another component. HOCs are common in third-party React libraries, such as Redux's `connect()`.

4.2.5 Reconciliation

Reconciliation[52] is the process by which React syncs changes in the application state to the DOM, so it:

- Reconstructs the virtual DOM
- Diffes the virtual DOM against the DOM
- Only makes the changes needed

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM.

5 React Native

A framework that relies on React for writing real, natively rendering mobile applications for iOS and Android using JavaScript. The fact that it actually renders using its host platform's standard rendering API enables it to stand out from most existing methods of cross-platform application development.

There are separate threads for UI, layout and JavaScript that communicate asynchronously through a “bridge”. The JavaScript thread can request UI elements to be shown and if is blocked, the UI will still work.

5.1 Internals

5.1.1 *Threading Model*

There are two important threads running in each React Native application. One of them is the main thread, which also runs in each standard native application. It handles displaying the elements of the user interface and processes user gestures. The other one is specific to React Native, its task is to execute the JavaScript code in a separate JavaScript engine. The JavaScript thread deals with the business logic of the application. It also defines the structure and the functionalities of the user interface. These two threads never communicate directly and never block each other. If the application accesses any native API, it is done on a separate native module thread. For example - Animations are handled in React Native by a separate native thread to offload the work from the JavaScript thread.

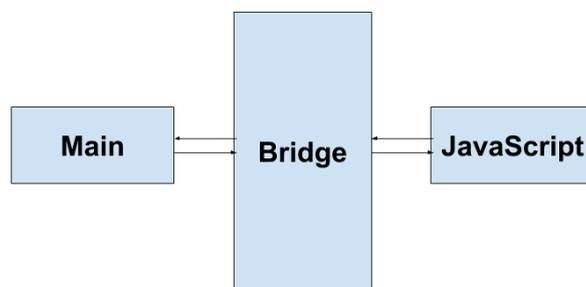


Illustration 5.1.1: The Threads and the Bridge

5.1.2 Bridge

Between the two threads there is the bridge, which is the core of React Native. The bridge has three important characteristics[53], it is:

- Asynchronous. It enables asynchronous communication between the threads. This ensures that they never block each other.
- Batched. It transfers messages from one thread to the other in an optimized way.
- Serializable. The two threads never share or operate with the same data. Instead, they exchange serialized messages.

5.1.3 JavaScript Environment

When using React Native, JavaScript code will run in two environments:

- In most cases, React Native will use JavaScriptCore, the JavaScript engine that powers Safari.
- When using Chrome debugging, all JavaScript code runs within Chrome itself, communicating with native code via WebSockets. Chrome uses V8 as its JavaScript engine.

While both environments are very similar, there may end up hitting some inconsistencies.

5.1.4 JavaScript Syntax Transformers and Pollyfills

React Native ships with the Babel JavaScript compiler. A full list of React Native's enabled transformations can be found in metro-react-native-babel-presets, for example. It provides ES5 and ES6 transformations like arrow functions, classes, spread operator, modules and async/await, template literals, etc. Also, specific transformations like JSX and Flow.

Many standard functions are also available on all the supported JavaScript runtimes by means of polyfills.

5.2 Differences with React Web

- Both uses JSX for defining components but React Native has a separate set of tags, some base components for defining user interface for mobile.
- React-Native is not made from web elements and can't be styled in the same way.
- No browser APIs: CSS animations, Canvas, SVG, etc., but some have been polyfilled (fetch, timers, console, etc.)
- Navigation: In a web browser, links to different pages are possible using the anchor (`<a>`) tag. When a user clicks on a link, the URL is pushed to the browser history stack. When the user presses the back button, the browser pops the item from the top of the history stack, so the active page is now the previously visited page. React Native doesn't have a built-in idea of a global history stack like a web browser does.
- Event handling: Unlike web, not every component has every interaction, there are only a few “touchable” components, like `Button` and `TouchableOpacity`. Web handlers will receive the event as an argument, but React Native handlers often receive different arguments.
- React Native's components are not globally in scope like React web components.
 - Import from 'react-native'
- The `div` tag in web corresponds to the `View` tag and the `span` tag to `Text` tag in React Native. All text must be wrapped by a `<Text />` tag.

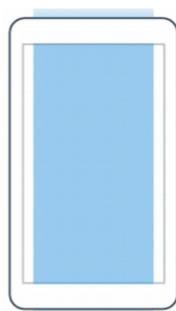
5.3 Components and APIs

React Native provides a number of built-in components, the following are the most important ones used in the application:

5.3.1 Basic Components

- *View*: The most fundamental component for building a UI. It is a container that supports layout with flexbox, style, and touch handling, and accessibility controls. View is designed to be nested inside other views and can have 0 to many children of any type.

- *Text*: A component for displaying text.
- *Image*: A component for displaying images. There are two ways to load the blob data, the first approach loads the image data from the network by passing an object with a *uri* property to source. The other way is using a local image file, by calling `require()` and passing the result to source.
- *TextInput*: A component for inputting text via a keyboard.
- *ScrollView*: Provides a scrolling container that can host multiple components and views. Is a special kind of View that has two parts, as shown in Illustration 5.3.1:
 - Container (the grey box), it's the outside View, its height can't exceed 100% of the window height



ScrollView

*Illustration
5.3.1:
ScrollView
parts*

- Content (marked in blue) is the inner part, it can be higher than the window height, it's what's moving inside the container.

ScrollView's style defines the outer container of the ScrollView, e.g its height and relations to siblings elements while `contentContainerStyle` defines the style of the inner container of it, e.g items alignments, padding, etc.

By default, ScrollView is laid out vertically. In order to scroll the content horizontally, the prop `horizontal` must set to true.

5.3.2 User Interface

- *Button*: A basic button component for handling touches that render accordingly on any platform. There are other components that allow a developer to customize “touchable” components.

5.3.3 List Views

React Native provides components for presenting lists of data, the most important is FlatList since it is used to show the user’s vineyards and the grid of pictures taken by a camera. There is also SectionList, a component with additional support for sections, it is used to display the Nodes and Cameras from a vineyard. Both of these components are performant. In detail:

5.3.3.1 FlatList

The Flatlist[54] component displays a scrolling list of changing, but similarly structured, data. It works well for long lists of data, where the number of items might change over time. Unlike the ScrollView, the it only renders elements that are currently showing on the screen, not all the elements at once. The component requires two props:

- *data*: source of information for the list. Array of data used to create the list, typically an array of objects.
- *renderItem*: takes one item from the source and returns a formatted component to render. function that will take an individual element of the data array and render a component for it.

Each item of the list must have a unique key, that allows the VirtualizedList (which is what FlatList is built on) to track items and aids in the efficiency of the list. The prop *keyExtractor* can be use to specify which piece of data should be used as the key.

Is fully cross-platform and provides:

- Optional horizontal mode.
- Configurable view-ability callbacks.
- Header support.
- Footer support.

- Separator support.
- Pull to Refresh.
- Scroll loading.
- ScrollToIndex support.

5.3.3.2 *SectionList*

`SectionList`[55] is like `FlatList` but with additional support for sections. It requires a `renderSectionHeader` function prop for section headers and instead of the `data` prop, it defines sections where each section:

- has its own data array
- can override the `renderItem` function with their own custom renderer

5.3.4 iOS/Android specific

DatePickerAndroid: Opens the standard Android date picker dialog.

The available keys for the options object are:

- `date` (Date object or timestamp in milliseconds): date to show by default
- `minDate` (Date or timestamp in milliseconds): minimum date that can be selected
- `maxDate` (Date object or timestamp in milliseconds): maximum date that can be selected
- `mode` (enum('calendar', 'spinner', 'default')): to set the date-picker mode to calendar/spinner/default
 - 'calendar': Show a date picker in calendar mode.
 - 'spinner': Show a date picker in spinner mode.
 - 'default': Show a default native date picker(spinner/calendar) based on android versions.

It provides the `open()` method, that returns a Promise which will be invoked an object containing `action`, `year`, `month` (0-11), `day` if the user picked a date.

This component is used to give the user the option to display the data of the sensor on that selected date.

5.3.5 Others

5.3.5.1 Animated

Animation can be defined as manipulating images or objects to appear as moving. React Native provides the Animated library for creating animations, it allows to declare a computation in JavaScript and compute it on the native thread and not on the JavaScript thread, so if the JavaScript thread is blocked, the animation will still run.

The steps are usually as follows:

1) Create a new Animated instance and define the starting value or the starting location of the animation in reference to the exact X, Y coordinates on the screen. The X, Y coordinates always start at the top-left corner of the screen.

2) Define the end value or ending location of the animation. Animated has built-in types to use to get from the starting location or value to our ending location or value. Each animation type provides a particular animation curve that controls how the values animate:

- Decay: starts with an initial velocity and gradually slows to a stop.
- Spring: provides a simple spring physics model.
- Timing: animates a value over time using easing functions.

3) Define which element to animate. React Native provides four components. These components do the binding of the animated values to the properties, and do targeted native updates to avoid the cost of the react render and reconciliation process on every frame.

- Animated.Image
- Animated.ScrollView
- Animated.Text
- Animated.View

Interpolation

`Interpolate()` is a method that is available to be called on any animated value, it interpolates the value before updating the property, e.g. mapping 0–1 to 0–10. Allows input ranges to map to different output ranges.

Handling gestures and other events

Gestures, like panning or scrolling, and other events can map directly to animated values using `Animated.event()`. The animated events help to extract the values from complex event objects.

5.3.6 Style

React Native uses JavaScript objects for styling, where object keys are based on CSS properties. Every component accepts a “style” prop where these are defined. The style prop can take an array of styles.

Additionally, React Native provides the `StyleSheet` functionally, which is the same as creating the objects for style, but have an additional optimization: only sends IDs over the bridge.

A component can specify the layout of its children using the Flexbox algorithm. Flexbox is designed to provide a consistent layout on different screen sizes.

Flexbox works the same way in React Native as it does in CSS on the web, with a few exceptions. The defaults are different, with `flexDirection` defaulting to column instead of row, and the flex parameter only supporting a single unit-less number.

To achieve the desired layout, Flexbox offers three main properties:

Property	Values	Description
flexDirection	'column', 'row'	Used to specify if elements will be aligned vertically or horizontally.
justifyContent	'center', 'flex-start', 'flex-end', 'space-around', 'space-between'	Used to determine how should elements be distributed inside the container.
alignItems	'center', 'flex-start', 'flex-end', 'stretched'	Used to determine how should elements be distributed inside the container along the secondary axis (opposite of flexDirection).

Table 5.3.1: Main properties of Flex

5.3.7 Direct Manipulation

It is sometimes necessary to make changes directly to a component without using state or props to trigger a re-render. *setNativeProps* is the React Native equivalent to setting properties directly on a DOM node.

5.3.8 Installation

5.3.8.1 Dependencies

Depending on the development OS and the target OS, the dependencies vary. For example, a Mac computer is required to build projects with native code for iOS. In this project the target OS is Android, so the requirements are:

- Node (8.3 or newer)
- Watchman (optional)
- The React Native Command Line Interface (React Native CLI)
- A JDK (JDK 8)
- Android Studio (or just the Command Line Tools)

Node comes with Node Packager Manager[56] (npm) which will be used to install not only React Native, but all of the application's dependencies. Instead of npm, yarn can also be used.

The first requirement when developing Android applications is an up-to-date version of JDK (Java SE Development Kit), at the moment JDK 8.

Android Studio installs the latest Android SDK by default. Building a React Native application with native code, however, requires a specific Android SDK, at the beginning of the project it was Android SDK Platform for API Level 23 (6.0 Marshmallow). Additional Android SDKs can be installed through the SDK Manager.

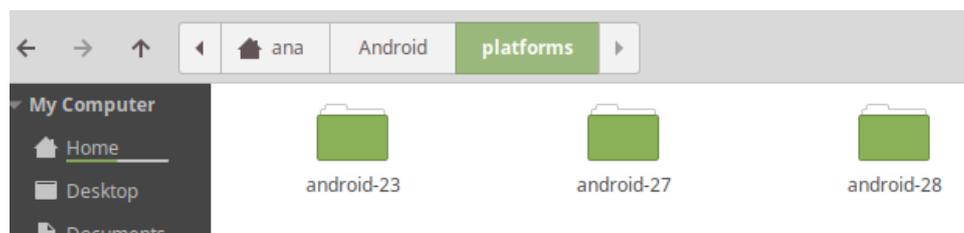


Illustration 5.3.2: Android platforms installed

Additionally, the developer must install:

- Google APIs
- Intel x86 Atom_64 System Image

It is important to set up the Android and Java environment variables. For this project, the versions of the dependencies installed are:

- Development OS: Linux Mint 19 Tara
- npm: 6.5.0
- Node: v8.11.3
- Yarn: 1.12.3
- Watchman: 4.9.0

5.3.9 React Native CLI Commands

To install the React Native CLI:

```
npm install -g react-native-cli
```

Listing 5.3-1: React Native CLI installation

This will create a new folder in the current directory and create a React Native project inside of it.

```
react-native init <project name>
```

Listing 5.3-2: Creation of a new React Native Project

As the Illustration 5.3.3 shows, the Android and iOS folders contain Objective-C and Java code for iOS and Android native parts. The node_modules directory contains all installed npm packages. In the root folder there are various configuration files for Babel, Flow[57], Git[58], Watchman and Yarn[59]. There is also an index JavaScript file, called index.js, that serves as an entry point of the application.

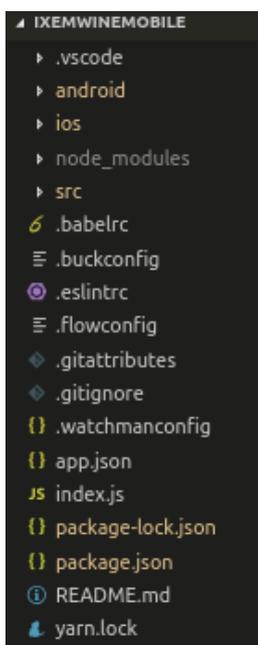


Illustration 5.3.3: Project structure

To run and build the application, in the case of Android/iOS case:

```
react-native run-android
```

Listing 5.3-3: Launch the application in an Android device or emulator

```
react-native run-ios
```

Listing 5.3-4: Launch the application in iOS device or emulator

The instructions build the native .app or .apk using the iOS or Android toolchains, starts the Metro Bundler, which minifies and serves the JSX and other assets such as images over to the device. On Android, it starts the adb server to push the .apk with all the native libraries included onto the device (with USB debugging enabled).

The react-native CLI will run the packager, which is in charge of bundling the JavaScript files and launch the emulator on the physical device.

The packager need to be running at all times while developing, the code changes will be reflected in the application by enabling hot reloading.

Metro server is configured to start on port 8081 by default. Once the application is launched in the simulator/device, a request is sent to the server for the bundle. The server then downloads all the required dependencies, bundles the JavaScript code and sends it back to the application. After this step, the application start working.

The development was done using a physical Android smartphone, running Android 8.0.0. To verify if the device is connected the command in Listing 5.3-5 can be used in the command line:

```
adb devices
```

Listing 5.3-5: Command to verify the connected smartphones

To run the application in a device, the developer must authorize the mobile device and enable USB debugging the following must be actions must be followed: “Settings > Developer Options > USB debugging”.

5.3.10 The CLI

The react-native-cli, installed by means of npm as a separate module, is a shell for accessing the CLI embedded in the React Native of each project. The commands and their effects are dependent on the version of the module of react-native in the context of the project.

Running react-native -help from inside a React Native project will list all of the current commands, as shown in Illustration 5.3.4:

```

ana@ana-UX32VD:~/code/thesis/ixemwinemobile$ react-native --version
react-native-cli: 2.0.1
react-native: 0.57.8
ana@ana-UX32VD:~/code/thesis/ixemwinemobile$ react-native --help
Usage: react-native [options] [command]

Options:
  -V, --version           output the version number
  -h, --help              output usage information

Commands:
  start [options]         starts the webservice
  run-ios [options]       builds your app and starts it on iOS simulator
  run-android [options]   builds your app and starts it on a connected Android emulator or device
  new-library [options]   generates a native library bridge
  bundle [options]        builds the javascript bundle for offline use
  unbundle [options]      Deprecated. Renamed to `ram-bundle`.
  ram-bundle [options]    builds javascript as a "Random Access Module" bundle for offline use
  eject [options]         Re-create the iOS and Android folders and native code
  link [options] [packageName] links all native dependencies (updates native build files)
  unlink [options] <packageName> unlink native dependency

```

Illustration 5.3.4: Options for commands available in the current Project

5.3.11 Debugging

From the device’s developer menu, it is possible to tap on “Debug JS Remotely.” This will launch Google Chrome and run the JavaScript in the browser instead of running it on the device. React Native will set up a WebSocket connection between the device and the browser that allows the developer to use Chrome’s developer console.

5.3.12 Linking libraries

Some React Native modules include native code for Android and/or iOS in addition to JavaScript. There are some extra steps to connect it with the native parts of the application.

In the case of Android, if there is the need to use native code, there are two ways to use Java with the Android SDK in a React Native project:

- put the Java code directly into the existing app by adding Java classes to `android/app/src/main/*` folder and use them from `MainActivity/MainApplication`.
- create an Android Library, which is the way any npm react-native packages (that go beyond JavaScript) work. The benefit is that it is reusable.

So, to link a library first install the component, using npm like a normal module, and then link it with the command listed in Listing 5.3-6:

```
react-native link <dependency-name>
```

Listing 5.3-6: Linking a native library

It is an automatic way for installing native dependencies and to avoid manually linking all the dependencies in the project. It works for both Android and iOS. However, the linking process of the libraries can vary, so it is important to check the installation steps for each of them.

6 Data Visualization

6.1 Scalable Vector Graphics (SVG)

SVG[60] is a language for describing 2D-graphics and graphical applications in XML which is then rendered using the SVG viewer. Most modern browsers support SVG and can display them as an images just like a regular JPG.

SVG drawings are created using a wide array of elements. For this project the following elements are used:

- `<svg>`: is a container that defines a new coordinate system and the viewing area where the SVG will be visible . It is used as the outermost element of any SVG document but it can also be used to embed a SVG fragment inside any SVG or HTML document.
- `<g>`: is a container used to group other SVG elements. Transformations applied to the `<g>` element are performed on all of its child elements, and any of its attributes are inherited by its child elements.
- `<line>`: basic shape used to create a line connecting two points.
- `<text>`: defines a graphics element consisting of text. It's possible to apply a gradient, pattern, clipping path, mask, or filter to `<text>`, just like any other SVG graphics element.
- `<path>`: generic element to define a shape. All the basic shapes can be created with a path element.

SVG support in React Native is via *react-native-svg*[61]. It provides SVG support to React Native on iOS and Android, and a compatibility layer for the web. This library contain native code and must be linked as explained in Linking libraries and reading the instruction provided in the library's instructions.

The library *svg-path-properties*[62] is a pure Javascript library used to obtain the functions `getPointAtLength(t)` and `getTotalLength()` needed to do some calculations for plotting the chart.

6.2 D3.js

D3.js[63] is a JavaScript library that exploits all the benefits provided by the DOM to visualize data using HTML, CSS and SVG. D3 manages the complexities of web standards and provides capabilities to browsers by combining powerful visualization components along with a data-driven approach. The library consists of many useful features including scaling, transformations, axes creation and many others.

D3, in its 4th version, is presented as a collection of modules developers can use independently, with minimal dependency between them, all neatly isolated in their own repository. The following modules are used in this project.

6.2.1 D3-scales

D3-scales[64] provides the encodings that map abstract data to visual representation. Every dataset has values within a domain and though the domains can vary drastically, one thing remains constant; the number of pixels available on the screen. These different domains needs to be mapped onto this output range. This is handled by D3-Scales property that maps the input domain to the output range. Once D3 scale function is defined by providing it with input domain and output range of pixels, the scale function can be called by passing the input value and it returns a scaled output value. D3 provides different types of scales such as linear, ordinal, logarithmic, square root. In the project, only the linear scale is used.

6.2.2 D3-shape

D3-shape[65] provides graphical primitives for visualization, such as lines and areas. This module supply a variety of shape generators. As with other aspects of D3, these shapes are driven by data: each shape generator exposes accessors that control how the input data are mapped to a visual representation.

6.2.3 D3-array

D3-array[66] provides array manipulation, ordering, searching, summarizing, among other things. Data in JavaScript is often represented by an iterable (such as an array, set or generator), and so iterable manipulation is a common task when analyzing or visualizing data.

7 Implementation

The development of the application was using an incremental approach, each functionality was implemented and tested incrementally until the application covered the most important use cases.

Two important matters are the Application State and the Application Navigation, those topics are described in the following chapters.

Another topic is the interaction of the application with the server through its API, covered in Web Services.

7.1 Actors of the system

The iXemWine is a community formed by the a network of public sensors in vineyards located in municipalities or farms distributed throughout Italy. Anyone can register and have access to it.

A user represents someone who can log into the iXemWine application. In addition to the basic information, has credentials that enables them to log in to the system. Each user has roles, assigned to them, that defines permissions to perform a group of tasks.

An user administrator can perform additional actions, like adding a camera sensor to a vineyard.

A vineyard is represented as an object with a unique identification and a name. It has information associated with it, like the its creation date and the city it is located in, in addition to the latitude and longitude.

A vineyard can have nodes and cameras associated with it. A node has sensors, a type, a alias, creation date and other configuration parameters. Each node's sensor has a channel in which it handles the actual monitoring data.

A user can own one, several or any vineyards, as shown in Illustration 7.2.1. In addition, a vineyard can be shared with other users. Both the shared and owned vineyards must be displayed in the application. In the case of a vineyard that is not owned and nor shared with the user, then only a limited view of that vineyard is shown.

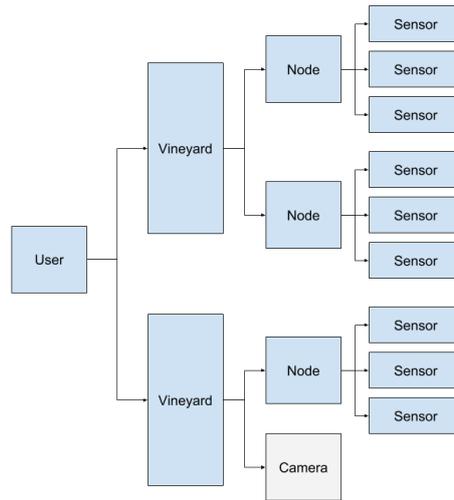


Illustration 7.2.1: Representation of a user's data

7.2 Functionality

The main functionalities of the application are described in the Illustration 7.2.2 and then described in detail in the following sections.

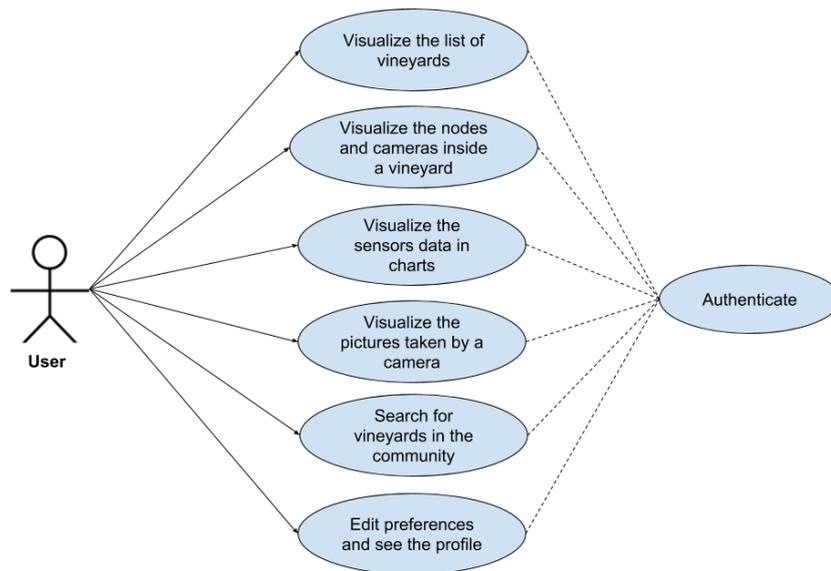


Illustration 7.2.2: Use Cases

7.2.1 Authenticate

The first screen of the application is where the user enters its credentials to login to the application. Once the user press the login button, it sends a HTTP POST

request to the iXemWine backend Authentication Server with the credentials and as a response it sends the token that will be used by the user on all the subsequent requests.

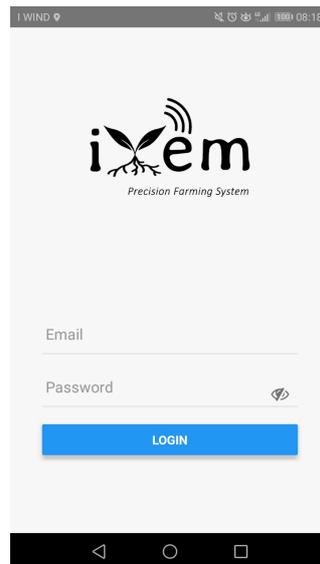
The only complication when developing this screen was that when the user pressed to type the username and password, the virtual keyboard showed up and covered text inputs fields.

For this reason, React Native provides the built-in `KeyboardAvoidingView` component but the documentation is vague, e.g., *“Android and iOS both interact with this prop differently. Android may behave better when given no behavior prop at all, whereas iOS is the opposite”*. [67]

To avoid the use of a third-party libraries and also to have a cross-platform solution, an alternative solution was reached using:

- the `Keyboard` module to control keyboard events [68]. `Keyboard` is a built-in React Native component that provide a couple of methods to listen for the virtual keyboard showing and hiding.
- animations, specifically `transform/translateY` to move the elements of the view vertically when the virtual keyboard shows/hide.
- the `Dimensions`[69] built-in API to obtain the width and height of the screen
- the currently focused field (and thus its position on the screen) in order to calculate the value to translate vertically the elements.

The result is the login screen, shown in Illustration 7.2.3:



*Illustration 7.2.3:
LoginScreen:
Authentication of the
user*

7.2.2 Visualize the list of Vineyards

To visualize the list of vineyards a user owns or has been shared with, a component named List was created, it is based on the FlatList, built-in component previously discussed.

The List component displays a Row component for each item. A Row component has a title, details and the function callback when an item is selected. When the user presses on one of the items, it navigates to another screen that shows the detail of that vineyard.

The list of vineyards a user owns and the ones shared with him are obtained once the user successfully login to the application and redirected to the first screen, which is the home.

In the Illustration 7.2.4 the Home screen is shown, it displays a Panel component that shows the number of vineyard owned, the shared ones and the total number of nodes of the user.



*Illustration 7.2.4:
Visualization of the list
of vineyards*

The other place where the List component is shown is in the tab Shared (“Condivisi”) but, in this case it only displays the List component.

7.2.3 Visualize the Nodes and Cameras inside a Vineyard

As stated previously, a vineyard can have nodes and cameras and to show them, the built-in SectionList component is used. The difference to FlatList is that it allows to separate the data in sections. Each section has its data source, its render function and a title.

In this case, the Camera and Nodes are two sections, as shown in Illustration 7.2.5 and the header of the screen is the vineyard’s name. When the user selects one of them it navigates to the specific node or camera.



*Illustration 7.2.5:
Visualization of sensors
and cameras in a
vineyard*

7.2.4 Visualize the Sensors data in Charts

Sensors data are visualized by means of a line charts. The chart configuration depends on the sensor's type, because in most of the cases the chart can have more than one data set to allow the user to contrast two physical quantities, like temperature and the dew point[70], across time.

There weren't many alternatives, it was hard to find a suitable library that met the functional and design needs at the same time. Only one library met the requirements, victory-native[71] but unfortunately it had known performance issue[72]. So, for this functionality there was a time overhead of learning and implementing the chart from scratch.

To build the chart, D3 was used. The first thing to do with the data set is to scale (using d3-scales) the x and y domain to the width and height of the chart, and then used the curve function, from the d3-shape library, to have a linear curve for the points.

The screen of the smartphones are small, so the proposed solution was to plot the chart without using the "traditional" scrolling, instead, a ScrollView component was

used on top of the chart so the user can slide a cursor over the line and to show a sliding label that acts as a x-axis.

To get the length of the line, the function `getTotalLength` of the `svg-path-properties` library was used.

The ticks of the y-axis can be fixed and be passed as a prop or they can be variable and change according to the data set.

The lines are plotted by means of the SVG path element, while the y-axis use the text element for showing the tick value and the line element for plotting the dashed lines.

To be able to slide throughout the chart line, the only consideration is that the `ScrollView`'s width attribute needs to be greater than the width of the screen and at least the size of the line length, since the "scrolling" is what enables to move from one end of the line to the other.

Another thing is listening to the `onScroll` event to get the scrolling value at any point, and with that value get the *x* and *y* coordinates (using `svg-path-properties`'s `getPointAtLength` function), and move the cursor along the graph using `setNativeProps` in order to do it without triggering a new rendering.

To allow the sliding movement of the label that shows the values at that specific point, an interpolation is performed where the input range is the selected line length and output range is the width of the screen (minus the label width). A `TextInput` component is used to be able to update the label value using again the `setNativeProps` method. To display the proper label value at a particular point the function `scale.invert`, from `D3-scales`, is used to revert from the *y* and *x* coordinates values to the original values.

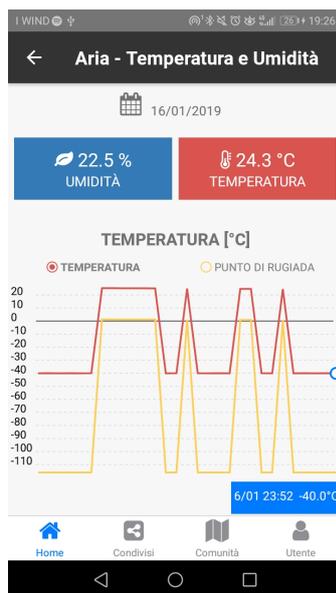
To be able to plot the two data sets in the same chart, in the chart component's state is stored a `selectedLine`, which is the line selected at a particular moment.

The chart component receives the mandatory dataset and the secondary one, the selection to where to place the cursor between them is performed by a boolean called `selectedMandatory`. When that boolean prop is true, the mandatory `lineLength` is selected otherwise the secondary is. If there is no secondary provided, there can not be any toggling.

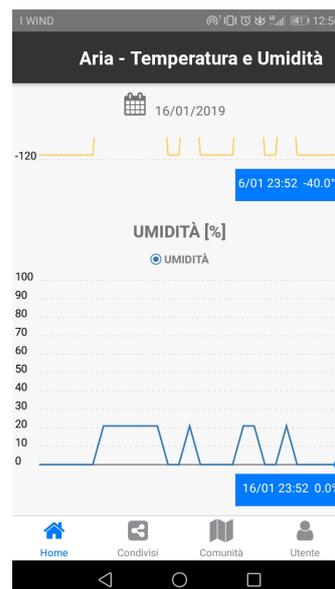
The chart component is encapsulated within a component called `ChartWithLabels` that add the Label functionality, showing a label for each data set, and the toggling between them represented as switching a radio button.

One more level of abstraction is created for each type of node, since each Node type has different icons, colors and constants.

The figure Illustration 7.2.6 and Illustration 7.2.7 shows an example of this functionality where the node type is a Temperature-Humidity sensor, so there are two charts, one for the temperature versus the dew point and the other with just the humidity:



*Illustration 7.2.6:
Visualization of the
temperature and the
dew point of a
Temperature-Humidity
Sensor*



*Illustration 7.2.7:
Visualization of the
humidity of a
Temperature-Humidity
Sensor*

Another type of sensor is the Two Layer Leaf Wetness sensor, that shows a top and a bottom humidity, but in this case to visualize the data only one chart is needed since the two data set refer to the same scale, as the Illustration 7.2.8 below shows:



*Illustration 7.2.8:
Visualization of a Two
Layer Leaf Wetness
Sensor*

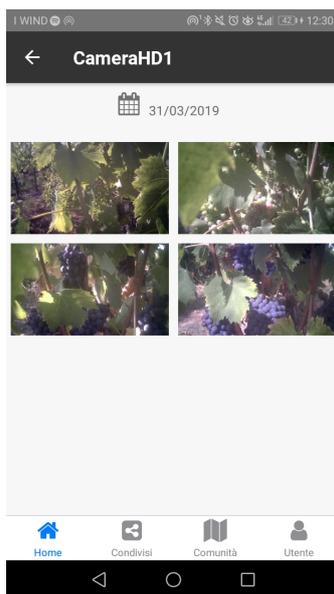
In addition to the chart, at the top of the screen, there is a Panel that shows the latest corresponding sensor's value acquired.

7.2.5 Visualize the pictures taken by a Camera

The pictures taken by the camera are shown in a grid, Illustration 7.2.9, sort of Gallery using a FlatList component that is set to render the pictures in columns.

The backend returns an array of objects that contain the URL of the pictures taken by the particular camera on the date selected. The pictures shown at the start by default are the ones taken that particular day.

The user can see the details of a picture by selecting it, which navigates to a different screen where the user can zoom-in and zoom-out the picture to see it in detail.



*Illustration 7.2.9:
Photo grid
visualization of the
pictures taken by a
camera*

7.2.6 Search for vineyards in the Community

The search for vineyards is by means of a map, as shown in Illustration 7.2.10, the positions of the vineyards are displayed as markers.

The only reliable way to use maps in React Native applications is to install the third-party *react-native-maps*[73] package.

The `MapView` component from that package is the main tool used and provides a declarative approach to control features on the map where they are specified as children of a `MapView`.

Each vineyard is represented by means of a `Marker` that is rendered on top of the map. A callout is displayed with the vineyard's basic information when the user presses on the marker. Then, if the user press on the callout, the user is redirected to a screen, which shows information of the selected vineyard.

To add the functionality of maps clustering the library *react-native-maps-super-cluster*[74] is used. It provides a `ClusteredMapView` component that receives the same children as a `MapView` but add the extra functionality for cluster rendering.



*Illustration 7.2.10:
Search and
visualization of the
vineyard's position*

7.2.7 Edit preferences and see the profile

The user can visualize its profile and change some preferences of the application. In this case, the language. The available languages are english and italian. The library `i18n-js`[75] is used, a plain popular JavaScript library which supports features such as date/time localization, number localization, locale fallback, etc.

Internationalization (i18n for short) is the process of adapting an application to work with different languages. The internationalization is done by having two files, `en.json` and `it.json`, containing strings in a flat JSON format. Both language files will contain key-value pairs with the same keys at any point in time containing the translations of the headers, texts messages, errors, etc. The different screens will import those strings from one of these files depending on the current language selected.

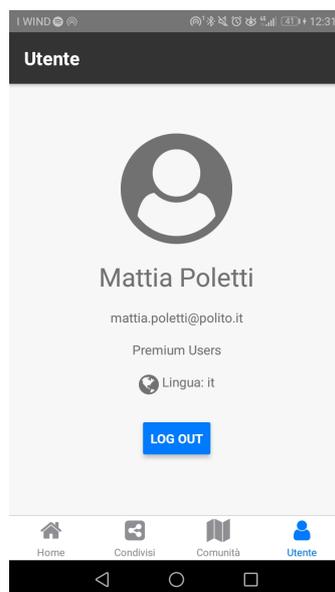
In order to configure some parameters, like the default language, the file `i18n.js` is created. Besides configuring i18n, it also exports a translate function and some other utility functions, for example to get or set the language.

To change the application's language there is a button in the User screen that shows the current language and when pressed it toggles the languages.

The language, stored in the Redux store, is always in sync with the language in the `i18n.js` file, and when the user changes the language an action is dispatched which changes the language in both `i18n` module and Redux store.

The navigator's header text and bottom tab labels have to be re-rendered on language change. The navigation library allows to pass props to the navigators, this props will be passed to each navigator instance and perform the re render when a language change occurs since it is connected to the Redux store.

The Illustration 7.2.11 shows the user's profile information and the functionality to change the language.



*Illustration 7.2.11:
User's profile and
language change screen*

More of Redux and navigation will be addressed on the following chapters.

8 Application Navigation

8.1 React Navigation

React Navigation[76] is used to move between screens in the application. It provides different types of navigation patterns, like a “stack of screens” or “tabs”. It’s a solution backed by the React Native community[77] that works with the native navigation components of both Android and iOS.

8.1.1 Navigator

A navigator is a component that implements a navigation pattern. Each navigator must have one or more routes. Each route must have a name and a screen component. The screen component is a React component that is rendered when the route is active.

A screen can also be another navigator, since they are components, so they can be nested. The result is that the application is a combination of these different navigators.

There are three types of built-in navigators used in the project are:

- SwitchNavigator: Only show one screen at a time. When navigate, it reset screen immediately without animation.
- StackNavigator: Contains screens as a stack. Each of the screens gets mounted only when navigating to that particular screen and gets un-mounted only when going back or manually reset the navigation state.
- TabNavigator: Contains tabs that the user can swipe. The tab screens get mounted all at once.

8.1.2 Authentication flow

Most applications require that a user authenticates in some way to have access to data associated with that user or other private content. Typically the flow looks like this:

- The user opens the application.

- The application loads some authentication state from persistent storage. The user is presented with either the authentication screen or the main application, depending on whether valid authentication state was loaded.
- When the user signs out, the authentication state is cleared and is sent back to authentication screens.

The purpose of SwitchNavigator is to only ever show one screen at a time. By default, it does not handle back actions and it resets routes to their default state when the user switch away. This is the exact behavior wanted from the authentication flow: when the users sign in, the state of the authentication flow is thrown away and unmount all of the screens, and the user press the hardware back button he can not go back to the authentication flow.

To switch between routes in the SwitchNavigator, the navigate action can be used.

The initial route name is set to be the screen that will fetch the user's authentication state from the persistent storage.

8.1.3 App containers

Containers are responsible for managing the application's state and linking the top-level navigator to the application environment. It must be created in the root of the application and use it to wrap the root navigator.

The app container, is a higher-order-component that maintains the navigation state of the application and handles interacting with the outside world to turn linking events into navigation actions and so on.

8.1.4 Navigation Prop

This prop will be passed into all screens, and it provides the following:

- `dispatch()`, will send an action up to the router
- `state`, is the current route for the screen
- `getParam()`, is a helper to access a parameter that may be on the route
- `navigate()`, used to navigate to another screen. It allows to pass parameters as well.

8.1.5 Passing parameters to routes

After creating a stack navigator with some routes, it is possible to navigate between those routes and to pass data to routes when navigating to them.

There are two pieces to this:

1. Pass parameters to a route by putting them in an object as a second parameter to the `navigation.navigate` function:

```
this.props.navigation.navigate('RouteName', { /*params go here*/ })
```

Listing 8.1-1: Navigate to a another screen and pass parameters

2. Read the parameters in the screen component:

```
this.props.navigation.getParam(paramName, defaultValue)
```

Listing 8.1-2: Read a parameter

8.1.6 Handling state

`ScreenProps` allows to provide any data to all the screens, in this case to pass state data down to the screen components. This is done in the top level navigator, which is also connected with `Redux`, so any changes in the state will cause a change of props pass to the screens as well.

So, if a prop is passed to the navigation component, it's accessible via the `screenProps` property and if a value is passed to the screen via `navigator.navigate()`, it's accessible by calling `navigator.getParam()`.

In this project, the only props that is passed is the `Language` since the UI (header and tab labels) needs to be updated dynamically when the user changes the language.

8.1.7 Navigation lifecycle

Consider a stack navigator with screens A and B. After navigating to A, its `componentDidMount` is called. When pushing B, its `componentDidMount` is also called, but A remains mounted on the stack and its `componentWillUnmount` is therefore not called. When going back from B to A, `componentWillUnmount` of B is

called, but `componentDidMount` of A is not because A remained mounted the whole time.

When, in combination with other navigators, like in this project where there is a Tab navigator with four tabs, where each tab is a Stack navigator. The initial screen is set to the `HomeScreen` and if the user navigate to a `VineyardScreen`. Then the user use the tab bar to switch to the `SharedScreens` and navigate to a shared `VineyardScreen`. After this sequence of operations is done, all 4 of the screens are mounted. If the user use the tab bar to switch back to the `HomeStack`, the user will be presented with the screen he left on before switching tabs, so the navigation state of the `HomeStack` has been preserved.

8.1.8 Application navigation

The application's navigation is accomplished by nesting different types of navigators. As the Illustration 8.1.1 shows, the `SwitchNavigator` performs the authentication flow.

Once the user is logged in, a `Bottom Tab Navigator` show four the tabs available: Home, Shared, Community and User. Each of them is a `Stack Navigator`.

Each tab being a navigator, allow the user to navigate to the different screens, for example in the case of the first tab it starts with the list of vineyards, then to the list of nodes and cameras that are inside the vineyard and finally arriving to the the particular sensor or camera and see the data displayed for a particular day.

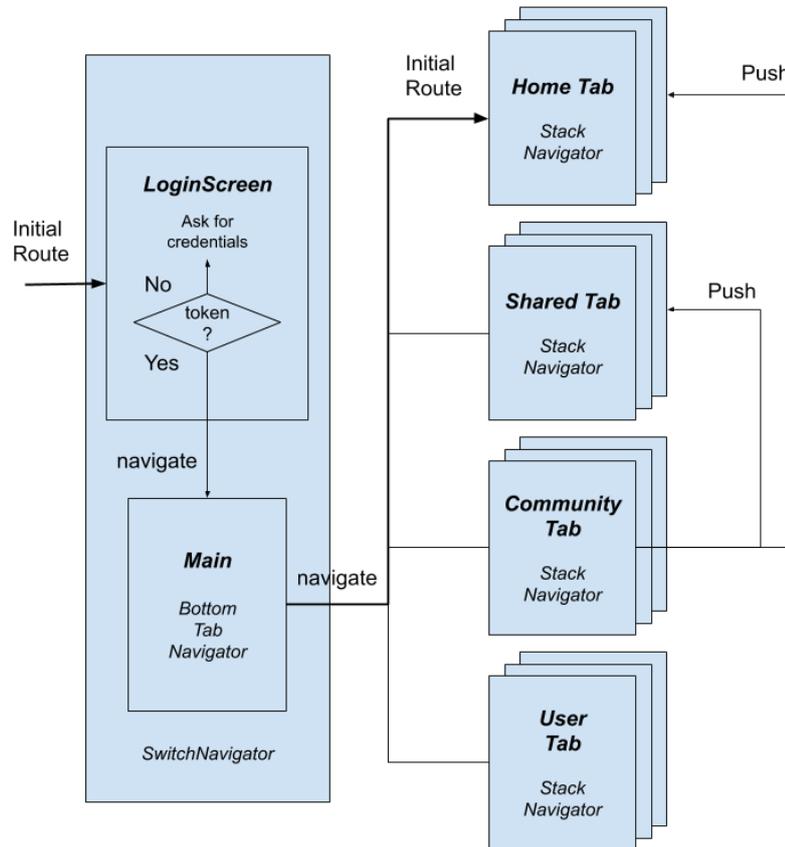


Illustration 8.1.1: Application Navigation

In the case of the Community tab, the navigation actions depend on the selected vineyard. If the selected vineyard :

- is owned by the user, push a new screen to Home tab.
- is shared with the user, push a new screen to Shared tab.
- is one from the Community, then it navigates to another screen in the same stack allowing the user only a partial view of that vineyard.

9 Application State

State is an important concept in any React application because it controls what the user can see and interact with.

Various pieces of state are persisted for different amounts of time and can be categorized into:

- **Short:** data that will change rapidly, for example the characters that a user types in an input text field. This type of data can be handled using the component's state.
- **Medium:** data that has to persist throughout the user navigation of the application, for example, the data returned from the server needs to be stored and will be used by the different screens. If that data is stored in some global location, it will be easier to access it. Such type of use cases fits Redux.
- **Long:** data that should be persisted when the user closes and re opens the application. This type of data should be stored somewhere else, for example the AsyncStorage provided by React Native.

9.1 Flux

In order to scale complexity and control the data flow, Facebook created an information architecture called Flux[78]. Thought for React, it utilizes a unidirectional data flow, where:

- the views react to changes in some number of “stores”.
- the only thing that can update data in a store is a “dispatcher”
- the only way to trigger the dispatcher is by invoking “actions”
- actions are triggered from the views

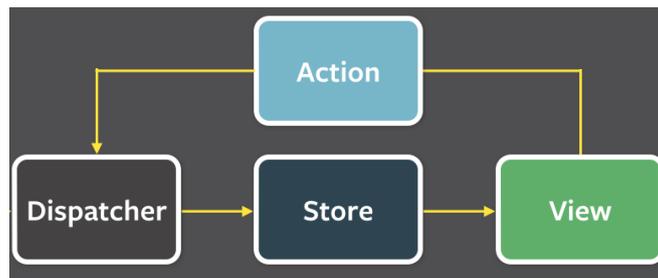


Illustration 9.1.1: Flux architecture

From the Flux documentation[link], Flux applications have three major parts: the dispatcher, the stores, and the views (React components). The store is an abstract concept that holds application state. The actions are simple objects containing the new data and an identifying *type* property.

The dispatcher is the central hub that manages all data flow in a Flux application. It is essentially a registry of callbacks into the stores and has no real intelligence of its own — it is a simple mechanism for distributing the actions to the stores. Each store registers itself and provides a callback. When an action creator provides the dispatcher with a new action, all stores in the application receive the action via the callbacks in the registry.

When a user interacts with a React view, the view propagates an action through a central dispatcher, to the various stores that hold the application's data and business logic, which updates all of the views that are affected.

9.2 Redux

Redux[79] is a data management library inspired by Flux, the only difference is that it only has a single source of truth for data. Presented as “*a predictable state container for JavaScript apps. It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience.*”[80]

The classic MVC (model-view-controller) behavior, in large-scale applications, has issues: the flow of data is bidirectional, which means that one change (a user input or API response) can affect the state of an application and many places in the UI and that can be hard to maintain and debug. The ability to represent the entire application state in a single object simplifies the developer experience since it becomes easier to think through the application flow, predict the outcome of new actions, and debug issues produced by any given action.

Redux flow is shown in Illustration 9.2.1 and has the following core concepts:

- **State:** The application's state described as a plain object.
- **Actions:** Object that contains the information required to make a state update, usually objects with a type key and a payload containing the information to be updated. The functions that create them are called action creators. Actions must be dispatched in order to affect the state.
- **Reducer:** function to tie state and actions together. The reducer takes the previous state and the action, then it applies the update. It should be a pure function, a function that doesn't alter input data, doesn't depend on external state and consistently provides the same output for the same input. So its result is deterministic and determined exclusively by arguments and also it must not have any side effects. It should be immutable, which means that it always returns a new state object.
- **Store:** The store contains the global state for the entire application. It is responsible for maintaining the state. The store exposes a getter via `getState()`, can only be updated by using `dispatch()` and can add listeners that get invoked when state changes.

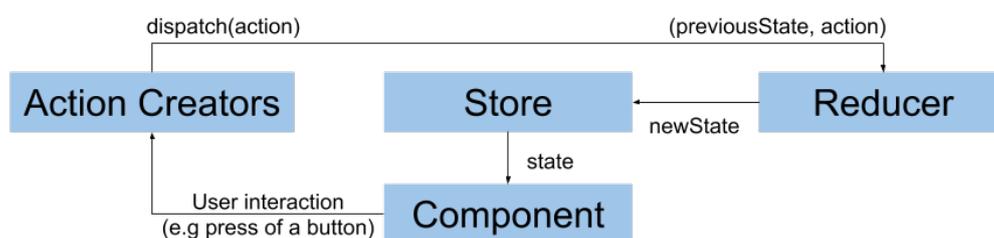


Illustration 9.2.1: Redux flow

9.2.1 Combining Reducers

As the application grows more complex, it is important to split the reducing function into separate functions, each managing independent parts of the state. The `combineReducers()`[81] helper function turns an object whose values are the different reducing functions into a single reducing function to pass to the `createStore`[82] method. The resulting reducer calls every child reducer, and gathers their results into

a single state object. The state produced by `combineReducers()` namespaces the states of each reducer under their keys as passed to `combineReducers()`.

9.2.2 React-redux

React-redux[83] This library has the official React's bindings for Redux. It lets React components read data from a Redux store, and dispatch actions to the store to update data, it does that using the following:

- `Connect()`: Higher-order component that helps subscribe to any subset of the store and bind the action creators. This function has two important arguments:
 - the `mapStateToProps` function: connects a part of the Redux state to the props of a React component, so it will have access to the exact part of the store it needs.
 - `mapDispatchToProps` function: connects the actions to the props of a React component, so the component will be able to dispatch actions.
- `Provider`: Gives children access to the Redux store. The `Provider` component is used to wrap the top-level component of the application. This will ensure that Redux store data is available to all the components.

9.2.3 Async Requests

A middleware is piece of code that sits between the actions and the reducers. Basically, takes the actions does something to it before passing it down to the reducer. This allows to extend Redux without having to touch the implementation.

Any function with this prototype can be middleware:

```
({getState, dispatch}) => next => action => void
```

Listing 9.2-1: Function prototype for a middleware

9.2.4 Redux Thunk

In order use Redux's synchronous action creators we defined earlier together with network requests is to use the Redux Thunk middleware[84]. It comes in a separate package called `redux-thunk`[85]. By using this specific middleware, an action creator

can return a function instead of an action object. This way, the action creator becomes a thunk.

“A thunk is a function that wraps an expression to delay its evaluation” [86]

When an action creator returns a function, that function will get executed by the Redux Thunk middleware. This function doesn't need to be pure; it is thus allowed to have side effects, including executing asynchronous API calls. The function can also dispatch actions—like those synchronous actions we defined earlier.

9.2.5 Persisting State

Redux-Persist[87] takes the Redux's state object and saves it to persisted storage. Then, when the application launches it retrieves this persisted state and saves it back to Redux.

The steps are as following according to the *The Definitive Guide to Redux Persist* [88]: when creating the Redux store, pass to the `createStore` function a `persistReducer` that wraps the application's root reducer. Once the store is created, pass it to the `persistStore` function, which ensures the Redux state is saved to persisted storage whenever it changes. At last, wrap the root component with `PersistGate` which delays the rendering of the application's UI until the persisted state has been retrieved and saved to Redux.

9.2.6 Application State

In the following Illustration 9.2.2, the state are represented by the rectangles and inside each of them there is the initial state objects. As explain in section 9.2.1, the state is namespaced.

The Cameras and Node are linked by the id stored inside each Vineyard.

The vineyards, the nodes and cameras are stored inside an objects called `myId` following the guide “Designing a Normalized State” in Redux's official documentation[89]. This state structure is much flatter overall.

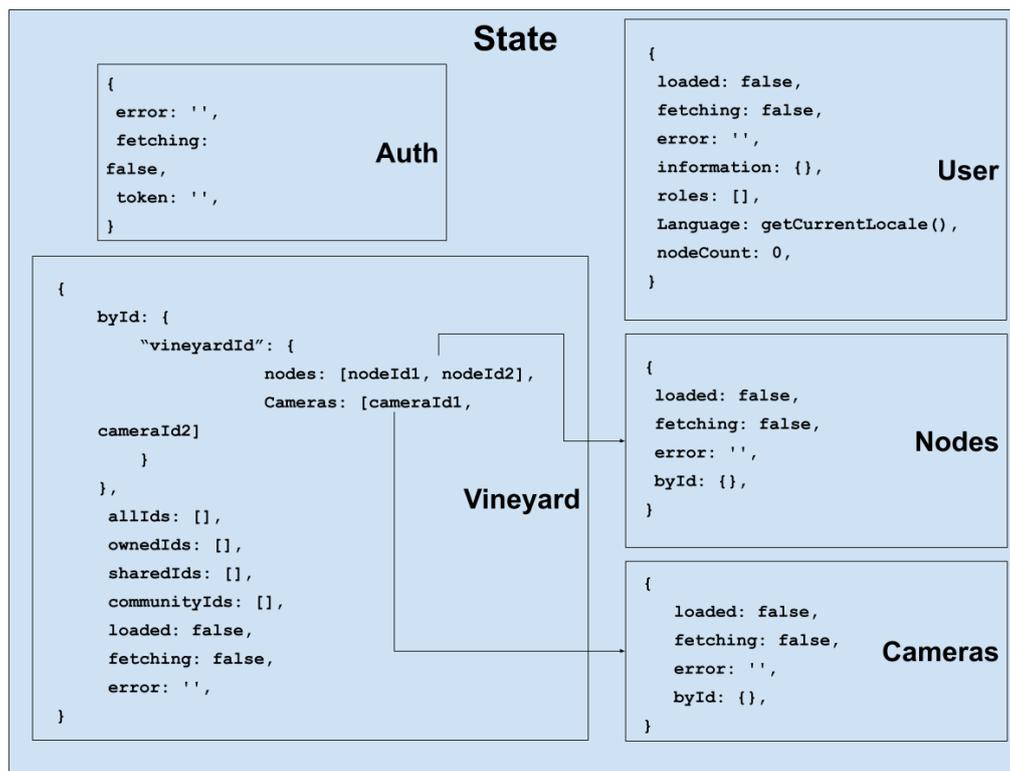


Illustration 9.2.2: Application State

In each part of the application that has to perform an asynchronous request in the state there are three values `fetching`, `loaded` and `error`.

The Illustration 9.2.3 shows the actions involved in each screen. Everything starts when the user logs in, after that it is redirected to the home screen, where the action `FETCH_USER_REQUEST` to get the user's data is dispatched. When the user's data is retrieved the reducers update the state and set the `loaded` flag to true to enable showing the data in the component. The nodes and Cameras are fetched by using the actions `FETCH_NODES_REQUEST` and `FETCH_CAMERAS_REQUEST`. When a user access a Node or Camera, the reducers add it in the correspondent reducer's state and also update the Node or Camera in the Vineyards object. The `FETCH_NODE_DATA_REQUEST` and `FETCH_CAMERA_DATA_REQUEST` fetch the specific sensor data for a particular day.

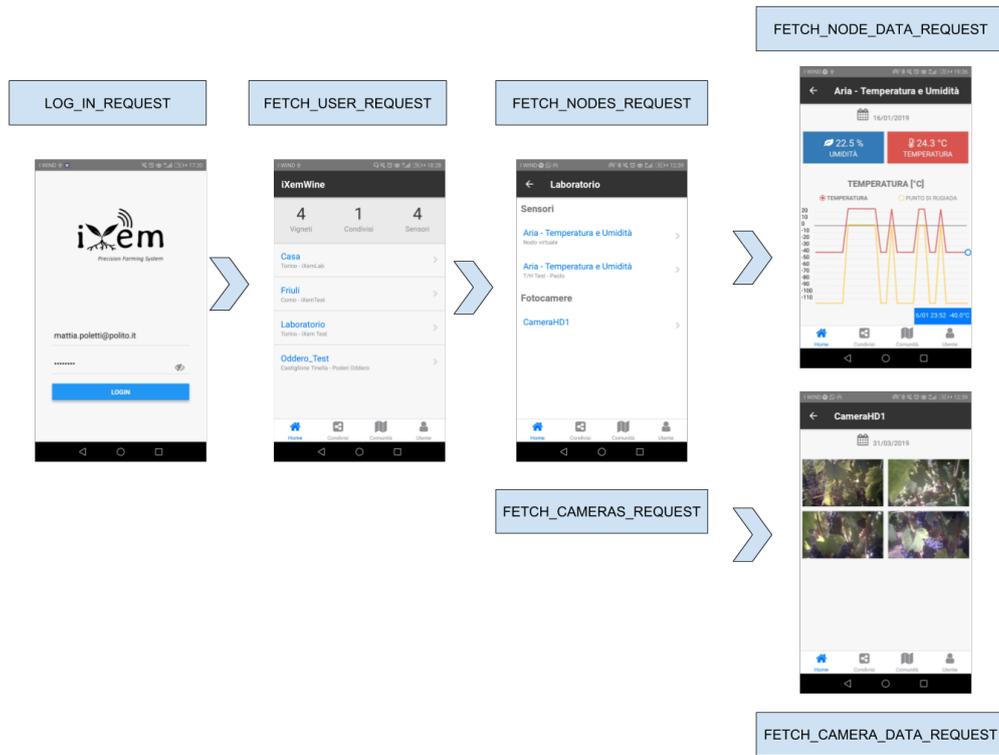


Illustration 9.2.3: Actions and Screens

10 Web Services

Any application that wants to rely on information not computed within itself needs to get it from somewhere and therefore, communicate with other resources using an API.

The user's information, its vineyards, the sensors and their data must be fetched from the server by making network requests.

The function to make request, `fetch()`, is polyfilled by React Native. That means that it is not natively part of JavaScript, but it is implemented to match the usage of the browser's `fetch()`. The function expects an URL and optionally some configuration. `Fetch()` returns a Promise, which is fulfilled with a Response object.

In React Native project it is possible to use the proposed ES2017 `async/await` syntax [90]. Both Promises and Async/Await described briefly below.

10.1.1 Promises:

- Allows writing asynchronous, non-blocking code
- Allows chaining callbacks and/or error handlers
 - `.then()` - executed after the previous Promise block returns
 - `.catch()` - executed if the previous Promise block errors

10.1.2 Async/Await

- Allows writing asynchronous code as if it were synchronous
 - Still non-blocking
- A function can be marked as `async`, and it will return a Promise
- Within an `async` function, it is possible to wait the value of another `async` function or Promise
- Use `try/catch` to handle errors

10.1.3 JSON Web Token

A JSON Web Token (JWT) is a JSON object that is defined in RFC 7519[91] as a safe way to represent a set of information between two parties. The token is composed of a:

- Header: Contains information about how the JWT signature should be computed.
- Payload: Stores data inside the JWT referred to as the “claims” of the JWT. In this case, the authentication server creates a JWT with the user information stored inside of it, for example user’s ID. There can be many claims. There are several different standard claims for the JWT payload, such as “iss” the issuer, “sub” the subject, and “exp” the expiration time. In this case, the “sub” claim stores the user ID needed to identify the user and fetch its data.
- Signature: The signature is computed using the following Listing 10-1:

```
data = base64urlEncode( header ) + “.” + base64urlEncode( payload )  
  
hashedData = hash( data, secret )  
  
signature = base64urlEncode( hashedData )
```

Listing 10-1: Pseudo code for calculating the signature

Since JWT are only signed and encoded, not encrypted, JWT do not guarantee any security for sensitive data.

As shown the Illustration 10.1[92], the user first send its credentials using a POST request, if they are correct, the server will provide the JWT. With the JWT, the user can then safely communicate with the application by sending it with each request.

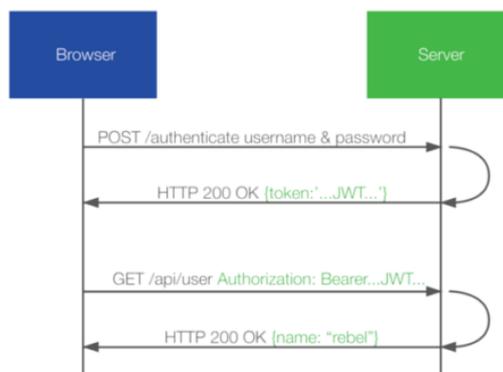


Illustration 10.1: Token-based Authentication

The library used to decode the token is *jwt-decode* [93]. A small browser library that helps decoding JWTs token which are Base64Url encoded.

10.1.4 Moment

The Moment.js library[94] is used for managing dates in JavaScript.

To get the current date and time:

```
const date = moment()
```

Listing 10-2: Getting current date and time

And to parse a date, a moment object can be initialized with a date by passing it a string:

```
const date = moment(string)
```

Listing 10-3: Initialization with specific date

It accepts any string, parsed according to (in order):

- ISO 8601[95]
- The RFC 2822 Date Time format[96]
- the formats accepted by the Date object[97]

10.1.5 Data Transformations

When asking for the sensors data, the time handling is important because. The when asking for a node's data it returns the stats, minimum, maximum and average values of the dataset, the values, the acquisition's timestamp in UTC[98] time using ISO-8601 and the timezone. So, using the timezone provides the acquisition time must be localized and for that is used *moment-timezones*[99], an add-on to the moment library.

11 Deploy

Android requires that all applications are digitally signed with a certificate before they can be installed, so to distribute the Android application via Google Play store, it is necessary two steps[100]:

- Generate a signed release APK
- Package the JavaScript bundle

11.1 Generating Signed APK

Generate a private signing key using `keytool`. This command prompts for passwords for the keystore and key and for the Distinguished Name fields for the key. It then generates the keystore as a file.

The file must be placed under the `android/app` directory. Some gradle variables must be set along with editing the application's gradle configuration to add the signing configuration.

11.2 Generate the release APK

Simply run the following in a terminal:

```
$ cd android
$ ./gradlew assembleRelease
```

Listing 11.2-1: Commands to generate the APK

Gradle's `assembleRelease` will bundle all the JavaScript needed to run the application into the APK. The generated APK can be found under `android/app/build/outputs/apk/release`.

12 Image Cloud Storage

Camera sensors allows the remote monitoring of the crop to check their status, show the growth of plant and store the history of it.

The cameras, located in the vineyards, are capable of taking high-resolution photos at regular intervals that are then sent using a GSM connection.

This second part of the thesis focus on adding the functionality of a image cloud storage and for that three alternatives where taken into account[101] of where to store the pictures.

12.1 Own cloud infrastructure(In-house-servers)

Benefits:

- Have physical control over the backup
- Keeps critical data in-house. No third party has access to the information
- No need to rely on an Internet connection for access to data
- Needed dedicated IT support.

Drawbacks:

- Requires a capital investment in hardware and infrastructure
- No up-time or recovery time guarantees.

12.2 Amazon cloud service

Benefits

- Storage can be added as needed. Solutions are often on-demand.
- Backup and restore can be initiated from anywhere.
- A service like Amazon S3[102] is not expensive.

Drawbacks

- Not owning the data.
- If the Internet goes down or the cloud provider's is down, there is no way to access to any of the information.

Between those options, the chosen one was Amazon Simple Storage Service, that will be discuss in the following sections. Basically the backend acts as a liaison between the users and cameras and the cloud, as depicted in Illustration 12.2.1:

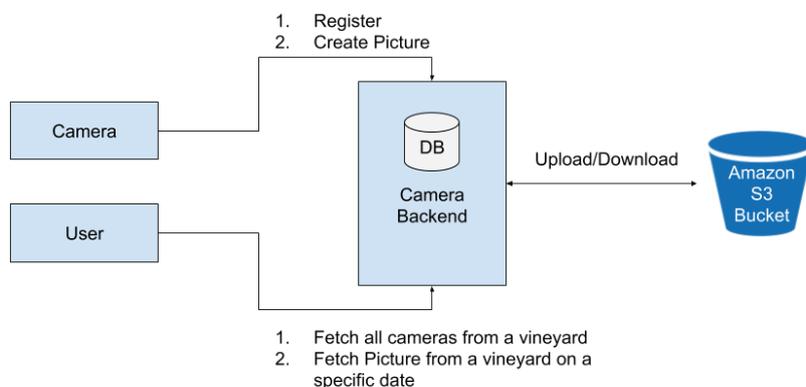


Illustration 12.2.1: Camera backend architecture

12.3 Functionalities to provide

The Table 12.3.1 show what the backend provides:

URL Endpoint	Function	HTTP Method	Data Provided	Token's scope check	Validate Signature with
/camera	Create a camera	POST	Alias, Vineyard ID, Latitude, Longitude	ADMIN	SECRET_HUMAN
/picture	Create a picture	POST	The picture	CAMERA	SECRET_M2M
/loginCamera	Get a camera token	POST	API key, Camera ID	-	-
/camera?	Get the	GET		USER	SECRET_

vineyard=<vineyardID>	list of cameras in a vineyard				HUMAN
/picture? camera=<cameraID> &month=<month> &year=<year> &day=<day>	Get the pictures taken by a camera in a specific day.	GET		USER	SECRET_HUMAN

Table 12.3.1: Backend functionalities

12.3.1 Authentication

The authentication of the users is still be handled by the iXemWine backend, which is the one in charge of giving the tokens to the users. Those tokens are generated with the constant SECRET_HUMAN.

The Camera backend need to know the SECRET_HUMAN in order to decode and validate the token received. If the token is valid, the next step is to check the scope of the user's decoded identity. If the user scope is an ADMIN, and the data needed to create a Camera exists, the new camera will be created in the data base, assigning it a random public ID and a API key.

Once the user ADMIN has created a Camera, it will provide the camera sensor with the API key and the public ID. When the Camera wants to send a picture it needs to obtain a camera token first, it must provide those two values in the body, If they are correct a token is returned for the camera.

That token then can be used when sending a picture, it has to have the CAMERA scope. The Camera's ID, extracted from the token when decoding, will be stored in the data base along with the generated ID, date and name of the picture.

For the last two endpoints, any regular user can use, are the ones that will be used in the mobile application to:

- obtain the list of Cameras that are in a specific vineyard
- obtain a list of URLs of the pictures taken by a particular camera on a particular day.

12.4 Flask

The web framework Flask[103] was chosen for this project because of its simplicity and some previous experience with the framework.

Flask is a framework for developing web applications in Python. Its goal is to be minimal without compromising functionality. It is extensible and flexible, so components like database and form validation can be chosen by the developer. The minimal nature of Flask makes it possible to write a web page in a very small amount of code. The following program, see Listing 12.4-1, creates a web server that serves a “Hello World!” page on the root.

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'
```

Listing 12.4-1: Minimal Flask application

The Flask framework is imported and a function is defined with the `app.route` decorator which tells the framework where to serve the page. The function returns the string “Hello World!” which is shown on the web page.

The following will present the Flask functionality that will be used in this part of the project.

Routing

The `app.route` decorator is used to bind a function to a URL. The parameter provided binds the function to the relative path.

HTTP methods

Another parameter for the `app.route` decorator is the methods allowed. This parameter enables other HTTP methods than the default GET. An example of enabling the POST method can be seen on Listing 12.4-2:

```
@app.route ('/login', methods =['GET', 'POST'])
def login ():
    if request.method == 'POST':
        perform_the_login()
    else :
        show_the_login_form()
```

Listing 12.4-2: Example of GET and POST method

Requests Interaction with the incoming request happens through the request object. This object contains all attributes of the request such as arguments from the query string, form data from POST requests and uploaded files.

Listing 12.4-3 shows a very simple use of the query string. The query string 'param1' is retrieved with the default value 'default' and assigned to param, which is then returned to the user

```
@app.route ( '/query_param')
def query_param():
    param = request.args.get( 'param1' , 'default')
    return param
```

Listing 12.4-3: Example of query string

The Response object

To get hold of the response before sending it to the client the `make_response()` method is used. `make_response()` returns a response object and takes a parameter for setting the status code, so creating the custom 404 handler just renders a template and sets the error code to 404. For example, the Listing 12.4-4 shows a 401 response to the client because the token was not present or can not be verified.

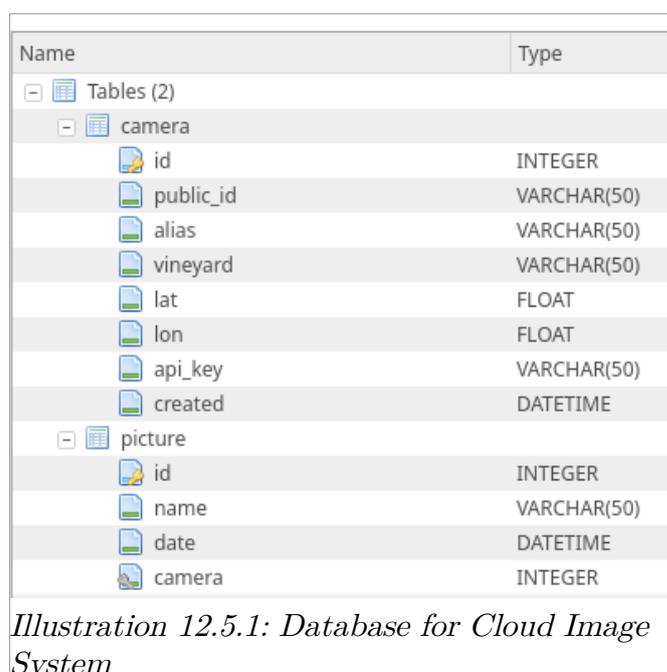
```
if not token:
    return make_response('Could not
        verify', 401, {'WWW-Authenticate':
            'Basic realm="Login required"'})
)
```

Listing 12.4-4: make_response example

12.5 SQLAlchemy

SQLAlchemy[104] is a toolkit for operating on SQL databases directly from Python. Its goal is to provide efficient and high performing database access, adapted into a simple and Pythonic domain language. The URI of the database is provided to the config attribute of the flask application, and a database object is created. This object contains a Model class that can be used to declare the model. In the project, there are two models is declared, Camera and Picture.

The created database is shown in Illustration 12.5.1:



Name	Type
Tables (2)	
camera	
id	INTEGER
public_id	VARCHAR(50)
alias	VARCHAR(50)
vineyard	VARCHAR(50)
lat	FLOAT
lon	FLOAT
api_key	VARCHAR(50)
created	DATETIME
picture	
id	INTEGER
name	VARCHAR(50)
date	DATETIME
camera	INTEGER

Illustration 12.5.1: Database for Cloud Image System

To populate the tables. When Cameras and Pictures are created, they are inserted into the database. And then to retrieve them it is possible to query the tables.

12.6 Amazon Simple Storage Service

Amazon S3 is an acronym for Amazon Simple Storage Service. It's a typical web service that lets users store and retrieve data in an object store via an API reachable over HTTPS.

The service offers unlimited storage space and stores the data in a highly available and durable way. Any kind of data can be stored, such as images, documents, and binaries, as long as the size of a single object doesn't exceed 5 TB. The user pay for

every GB stored in S3, and also incur in minor costs for every request and transferred data.

S3 uses buckets to group objects. A bucket is a container for objects with a globally unique name, the bucket name chosen can not be the used by any other AWS customer in any other region.

The user can access S3 via HTTPS using the Management Console, AWS Command Line Interface or depending the programming language the SDK, in this case Python (Boto3) to upload and download objects.

12.7 AWS Command Line Interface

The AWS Command Line Interface[105] (AWS CLI) is a unified tool that provides an interface for interacting with all parts of AWS. After the installation, can install via pip, run the command `aws configure` in the console and the CLI will prompt the necessary steps in order to fill in a few items including the keys and the region. When finished, the credentials will be stored in the home directory `.aws/credentials`.

12.8 Boto3

Given that the Camera backend uses Python, Boto3 is used to access the AWS features.

“Boto3 is the Amazon Web Services (AWS) SDK for Python. It enables Python developers to create, configure, and manage AWS services, such as EC2 and S3. Boto provides an easy to use, object-oriented API, as well as low-level access to AWS services”[106].

Installation is done via pip[107]. To interact with Amazon S3 through Boto3, the credentials must be set, as explained in the previous section using AWS CLI.

Creating Clients

Client[108] provide a low-level interface to AWS whose methods map close to 1:1 with service APIs. To construct a client to interact with S3, use the following Listing 12.8-1:

```
import boto3

# Create a low-level client with the service name
s3 = boto3.client('s3')
```

Listing 12.8-1: Create a client

After importing the boto3 library and constructing a client to interact with S3, the following two operations are needed:

12.8.1.1 Upload a File

To upload a file to an S3 object, the function `upload_file`[109] is provided. Usage:

```
s3.upload_file(Filename, Bucket, Key, ExtraArgs)
```

Listing 12.8-2: Upload a file

Parameters:

- `Filename` (str) -- The path to the file to upload.
- `Bucket` (str) -- The name of the bucket to upload to.
- `Key` (str) -- The name of the key to upload to.
- `ExtraArgs` (dict) -- Extra arguments that may be passed to the client operation.

12.8.1.2 Generate a pre-signed URL

A pre-signed S3 URL provides a secure, temporary access to objects in a S3 bucket. It provides temporary read access to the pictures stored in the bucket. It is obtained by means of the function `generate_presigned_url`[110]. The Listing 12.8-3 shows how to generate a pre-signed S3 URL that will allow the `GetObject` API call on the object:

```
url = s3.generate_presigned_url(  
    'get_object',  
    Params = {  
        'Bucket': 'bucket-name',  
        'Key': 'picture-name'  
    },  
    ExpiresIn = 3600  
)
```

Listing 12.8-3: Generate a presigned URL to return to the client

12.9 Proposed Solution

The proposed solution is to add a Camera Service backend to the already existing iXemWine backend, where each maintain a separate environment, dependencies and database.

The environment for the Camera Service includes the tools discussed in this chapter: Flask, SQLAlchemy, AWS CLI and Boto3.

The only thing they share is the secret keys in order to decode and verify the users. The iXemWine backend will still work as the default Authentication server for the clients.

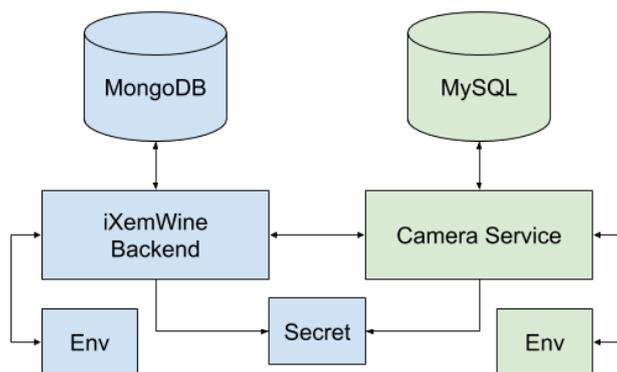


Illustration 12.9.1: Proposed Solution for Cloud Image System

13 Conclusion

The thesis engaged in creating a mobile application to allow the users to visualize the vineyard data in a meaningful way structuring the application using lists, seeing positions in the map, their sensors data in charts and pictures.

Firstly, the background for the project was examined, including important JavaScript and React concepts. The new acquired knowledge of React Native for the project was then described.

The application was implemented with the aim of being reusable, in the sense that developing a version for iOS would not require having to do major changes.

Regarding the second part, for the camera service some web concepts regarding Flask, SQLAlchemy and some Python were needed, because of the Boto3. The use of Amazon Web Services required some time to learn.

Overall it was a positive experience throughout the development of the application, it required a lot of new knowledge specially tools like Redux that have a learning curve and added some complexity.

13.1 Future work

It was covered only that which was of relevant to the app originally set out to create. Of course there are features that can be added in the future, for example:

- Having an iOS version of the application
- Create a notification system and set up remote push notifications
- Allow to to change other user's preference like theme

14 References

- [1] IoT Definition. https://en.wikipedia.org/wiki/Internet_of_things
- [2] WSN Definitions and its applications.
<https://www.elprocus.com/introduction-to-wireless-sensor-networks-types-and-applications/>
- [3] iXemWine. <https://ixem.wine/>
- [4] React Native. <https://facebook.github.io/react-native/>
- [5] JavaScript. <https://www.javascript.com/>
- [6] iOS. Apple's mobile operating system. <https://developer.apple.com/ios/>
- [7] Android. Google's mobile operating system. <https://www.android.com/>
- [8] LPWAN Definition. <https://en.wikipedia.org/wiki/LPWAN>
- [9] TCP/IP Protocol Suite.
https://en.wikipedia.org/wiki/Internet_protocol_suite
- [10] Python. <https://www.python.org/>
- [11] MQTT Protocol. <http://mqtt.org/>
- [12] Restful Software Architecture.
https://en.wikipedia.org/wiki/Representational_state_transfer
- [13] API definition.
https://en.wikipedia.org/wiki/Application_programming_interface
- [14] HTTP. <https://www.w3.org/Protocols/HTTP/1.1/rfc2616bis/draft-lafon-rfc2616bis-03.html>
- [15] Objective-C. <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html>
- [16] Swift. <https://docs.swift.org/>
- [17] Java. <https://www.java.com/>
- [18] Kotlin. <https://kotlinlang.org/>

-
- [19] PhoneGap. <https://phonegap.com/>
 - [20] Apache Cordova. <https://cordova.apache.org/>
 - [21] Ionic. <https://ionicframework.com/>
 - [22] Google Chrome. <https://www.google.com/chrome/>
 - [23] Node.js. <https://nodejs.org/>
 - [24] V8 JavaScript Engine. <https://v8.dev/>
 - [25] Mozilla Firefox. <https://www.mozilla.org/en-US/firefox/>
 - [26] SpiderMonkey.
<https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey>
 - [27] Safari. <https://www.apple.com/lae/safari/>
 - [28] JavaScriptCore.
<https://developer.apple.com/documentation/javascriptcore>
 - [29] Microsoft Edge. <https://www.microsoft.com/en-us/windows/microsoft-edge>
 - [30] Internet Explorer. <https://www.microsoft.com/en-us/download/internet-explorer.aspx>
 - [31] Chakra. <https://github.com/Microsoft/ChakraCore>
 - [32] ECMAScript® 2015 Language Specification. <https://www.ecma-international.org/ecma-262/6.0/>
 - [33] Kangax. <https://kangax.github.io/>
 - [34] ES6 compatibility table. <https://kangax.github.io/compat-table/es6/>
 - [35] Babel. <https://babeljs.io/>
 - [36] React. <https://reactjs.org/>
 - [37] JSX. <https://reactjs.org/docs/introducing-jsx.html>
 - [38] SVG. <https://www.w3.org/TR/SVG2/>
 - [39] Apache. <https://httpd.apache.org/>
 - [40] PHP. <https://php.net/>
 - [41] Metro. <https://facebook.github.io/metro/>

-
- [42] WebSocket. <https://developer.mozilla.org/en-US/docs/Web/API/WebSocket>
- [43] jest-haste-map. <https://github.com/facebook/jest/tree/master/packages/jest-haste-map>
- [44] Jest. <https://jestjs.io/>
- [45] Watchman. <https://facebook.github.io/watchman/>
- [46] fs.watch. <https://nodejs.org/docs/latest/api/fs.html>
- [47] DOM. Document Object Model (DOM) Technical Reports. <https://www.w3.org/DOM/DOMTR>
- [48] Harvard University.(2018). *Mobile App Development with React Native*. Produced by Jordan Hayashi and David J. Malan. <https://cs50.github.io/mobile/>
- [49] React lifecycle methods diagram. <http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>
- [50] Refs and the DOM. <https://reactjs.org/docs/refs-and-the-dom.html>
- [51] High Order Component. <https://reactjs.org/docs/higher-order-components.html>
- [52] Reconciliation. <https://reactjs.org/docs/reconciliation.html>
- [53] React Native: What it is and how it works. <https://medium.com/we-talk-it/react-native-what-it-is-and-how-it-works-e2182d008f5e>
- [54] FlatList. <https://facebook.github.io/react-native/docs/flatlist.html>
- [55] SectionList. <https://facebook.github.io/react-native/docs/sectionlist.html>
- [56] NPM. Node Packager Manager. <https://www.npmjs.com/>
- [57] Flow. <https://flow.org/en/docs/react/>
- [58] GIT. Open Source Distributed Version Control System. <https://git-scm.com/>
- [59] Yarn. Fast, reliable, and secure dependency management. <https://yarnpkg.com/>
- [60] SVG. <https://www.w3.org/Graphics/SVG/>

-
- [61] React-Native-SVG. <https://github.com/react-native-community/react-native-svg>
 - [62] Svg-path-properties. <https://github.com/rveciana/svg-path-properties>
 - [63] D3. Data-Driven Document. <https://d3js.org/>
 - [64] D3-scale. <https://github.com/d3/d3-scale>
 - [65] D3-shape . <https://github.com/d3/d3-shape>
 - [66] D3-array. <https://github.com/d3/d3-array>
 - [67] KeyboardAvoidingView.
<https://facebook.github.io/react-native/docs/keyboardavoidingview#behavior>
 - [68] Keyboard events. <https://facebook.github.io/react-native/docs/keyboard>
 - [69] Dimensions. <https://facebook.github.io/react-native/docs/dimensions.html>
 - [70] Dew Point Concept. https://en.wikipedia.org/wiki/Dew_point
 - [71] Victory Native. <https://github.com/FormidableLabs/victory-native>
 - [72] Victory Native performance issue.
<https://github.com/FormidableLabs/victory-native/issues/369>
 - [73] React-native-maps. <https://github.com/react-native-community/react-native-maps>
 - [74] React-native-maps-super-cluster. <https://github.com/novalabio/react-native-maps-super-cluster>
 - [75] i18n Translations for Javascript. <https://github.com/fnando/i18n-js>
 - [76] React Navigation. <https://reactnavigation.org/>
 - [77] React-native-community. <https://github.com/react-native-community>
 - [78] Flux, <https://facebook.github.io/flux/>
 - [79] Redux, <https://redux.js.org/>
 - [80] Redux, Getting Started with Redux,
<https://redux.js.org/introduction/getting-started>
 - [81] Combine Reducers. <https://redux.js.org/api/combinereducers>
 - [82] Create Store. <https://redux.js.org/api/createstore>

-
- [83] React-redux, Official React bindings for Redux. <https://react-redux.js.org/>
 - [84] Redux Middleware, <https://redux.js.org/advanced/middleware>
 - [85] Redux Thunk. <https://github.com/reduxjs/redux-thunk>
 - [86] What is a thunk?. <https://github.com/reduxjs/redux-thunk#whats-a-thunk>
 - [87] Redux-persist, <https://github.com/rt2zz/redux-persist>
 - [88] The Definitive Guide to Redux Persist, <https://blog.reactnativecoach.com/the-definitive-guide-to-redux-persist-84738167975>
 - [89] Normalizing State Shape. <https://redux.js.org/recipes/structuring-reducers/normalizing-state-shape>
 - [90] React Native. Network: fetch and async/await. <https://facebook.github.io/react-native/docs/network>
 - [91] JSON Web Tokens. <https://jwt.io/>
 - [92] Image for token based authentication. <https://stormpath.com/blog/token-authentication-scalable-user-mgmt>
 - [93] Jwt-decode library. <https://github.com/auth0/jwt-decode>
 - [94] Moment.js Documentation. <https://momentjs.com/docs/>
 - [95] ISO 8601. https://en.wikipedia.org/wiki/ISO_8601
 - [96] RFC 2822 Date Time Format. <https://tools.ietf.org/html/rfc2822#section-3.3>
 - [97] Date object. https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Date
 - [98] UTC. https://en.wikipedia.org/wiki/Coordinated_Universal_Time
 - [99] Moment Timezone. <https://momentjs.com/timezone/>
 - [100] Generating a signed APK. <https://facebook.github.io/react-native/docs/signed-apk-android>
 - [101] Cloud vs In-house-servers. <https://sysgen.ca/cloud-vs-in-house-servers/>

- [102] Amazon Simple Storage Service (Amazon S3).
<https://aws.amazon.com/s3/>
- [103] Flask. <http://flask.pocoo.org/>
- [104] SQL Alchemy. <https://www.sqlalchemy.org/>
- [105] AWS-CLI. <https://docs.aws.amazon.com/cli/index.html>
- [106] Boto 3 documentation. <https://boto3.amazonaws.com/v1/documentation/api/latest/index.html#>
- [107] Pip. Package-management System. <https://pypi.org/project/pip/>
- [108] Boto3 S3 client.
<https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#id201>
- [109] Boto3 `upload_file` function.
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.upload_file
- [110] Boto3 `generate_presigned_url`.
https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/s3.html#S3.Client.generate_presigned_url