Developer Manual

Sons Of SWE - Project Marvin

sonsofswe.swe@gmail.com

Informations about the manual

Version	2.0.0
Redaction	Cavallin Giovanni Panozzo Stefano
Verification	Favero Andrea
Approval	Thiella Eleonora
Use	External
Distribution	Vardanega Tullio Cardin Riccardo Red Babel Sons Of SWE group

Description

This document is the developer manual of Project Marvin

Diary of changes

Version	Date	Description	Author	Role
1.0.0	2018-06-07	Approval	Thiella Eleonora	*Project responsible*
0.2.0	2018-06-06	Verification	Andrea Favero	*Verifier*
0.1.1	2018-06-04	Frontend Written	Giovanni Cavallin	*Programmer*
0.1.0	2018-06-02	Backend written	Stefano Panozzo	*Verifier*
0.0.1	2018-05-31	Written the document skeleton	Giovanni Cavallin	*Programmer*

Contents

List of Figures

1 Introduction

This is the developer manual of **Marvin**, a Đapp ran on the EVM, that shapes a subset of Uniweb functionalities and based on the *Truffle* framework. Uniweb is the University of Padua's informative system. It allows students to keep track of their academic carrier. Professors use Uniweb to see the lists of students that are registered to their exams and, see the exams to which they have been assigned.
This manual is intended for programmers wishing to customize or extend **Marvin**. Those are expected to know and understand the *React-redux* as well as the *truffle* frameworks. In addition, the knowledge of the *solidity* language is mandatory to understand the database on which the application is based. A basic understanding of the Sass preprocessor CSS, while not required, is a plus.

1.1 What is in the Manual

This manual will cover most of the aspects of custom development and maintenance of Marvin. In particular it is divided into two main sections: *Frontend* and *Backend*. So if the reader wants to update or modify the `solidity` database part, he should skip all the *Frontend* part, instead if his purpose is to modify how the application is rendered on screen, then he should go to the first one.

2 Installation guide for local development

First of all you have to install Npm and Git. Check if they are working correctly typing in your terminal

```
$ node -v
$ npm -v
$ git --version
```

After that, if you're using Windows digit

```
$ npm install --global --production windows-build-tools
```

to install Python and other utilities that are necessary to make the demo works.

Download or clone the repository hosted at `https://github.com/SOS-SonsOfSwe/Marvin-SoS`.

Install MetaMask as an extension for a supported browser.

Go inside the repository folder (in case you aren't already there) and use npm to install other required programs typing the following commands:

```
$ npm install -g ganache-cli
$ npm install -g truffle
```

Type

```
$ npm install
```

Make executable the scripts *startBlockchain.ps1* and *loadProject.ps1* (if you're using an UNIX-like operating system you can type *chmod +x <nameOfTheFile>*).

Execute the script *startBlockchain.ps1* or otherwise type **ganache-cli -a 10 -m "candy maple cake sugar pudding cream honey rich smooth crumble sweet treat" -p 9545** to get the same result.

After that open a new terminal in the same folder and execute the script *loadProject.ps1*, otherwise to get the same result you can type the following commands:

```
$ truffle compile
$ truffle migrate
$ npm run start
```

At this point you will notice that your browser is automatically started and that it visualizes the webpage of Marvin running at the localhost address. Now you have to connect MetaMask: in your browser click on the MetaMask icon and accept the Privacy Notice and the Terms of use. Then click on **Main Network** and choose **Custom RPC**, type in the first form **http://localhost:9545** as in Figure 1 and click **Save**.

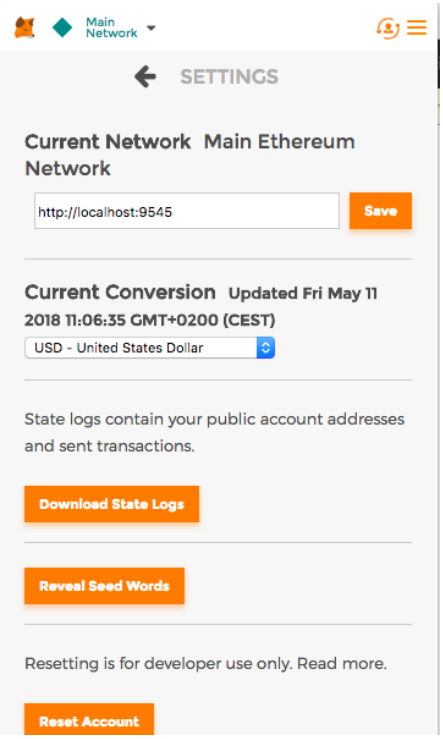
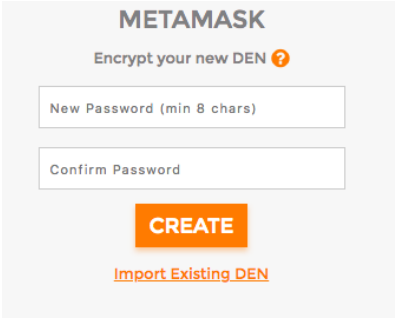
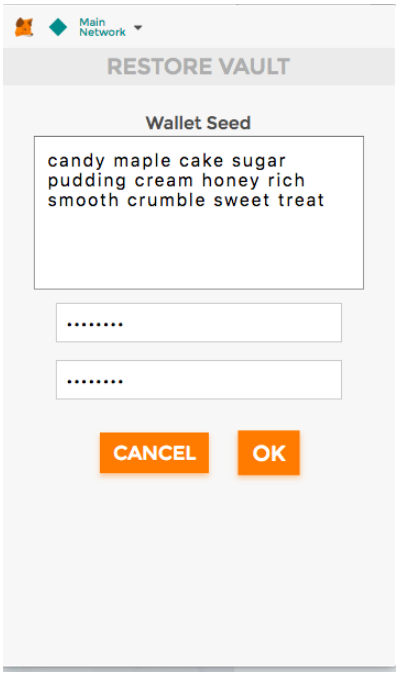


Figure 1: Set the RPC typing **http://localhost:9545**

Now as in Figure **??** click on **Import Existing DEN** and (as in Figure 2b) insert the seed phrase **candy maple cake sugar pudding cream honey rich smooth crumble sweet treat** and the password that you want to use for the account.



(a) Click **Import Existing DEN**



(b) Metamask DEN and seed phrase

Figure 2: main caption

Now you're ready to try the demo!

3 Project Folder

The project folder contains the following subfolders:

- `contracts`: it contains the blockchain contracts written in the Solidity programming language
- `migrations`: it contains javascript files that help you deploy contracts to the Ethereum network. These files are responsible for staging your deployment tasks, and they're written under the assumption that your deployment needs will change over time. As your project evolves, you'll create new migration scripts to further this evolution on the blockchain.
- `api`: it contains all the *adapters* to the external API such as `IPFS` and `web3`;
- `public`: it contains some media files, such as images, that can be accessible as external source;
- `scripts`: it contains files (build.js, start.js, test.js) written to compile and to create the local execution workspace for Marvin;
- `src`: it contains the part of the project related to the frontend development;
- `webpack`: it contains webpack, a static module bundler for JavaScript. It internally builds a dependency graph which maps every module needed by the project and generates one or more bundles

The folders `scripts` and `webpack` are responsible for getting the application start, test and build. If tyou are not confident with the settings of the *truffle* framework and the *webpack* package it's recommended to not modify those files.

For any suggestion please feel free to contact us or open an issue on the GitHub portal.

4 Architecture

In Figure 3 it is shown the general diagram of the packages

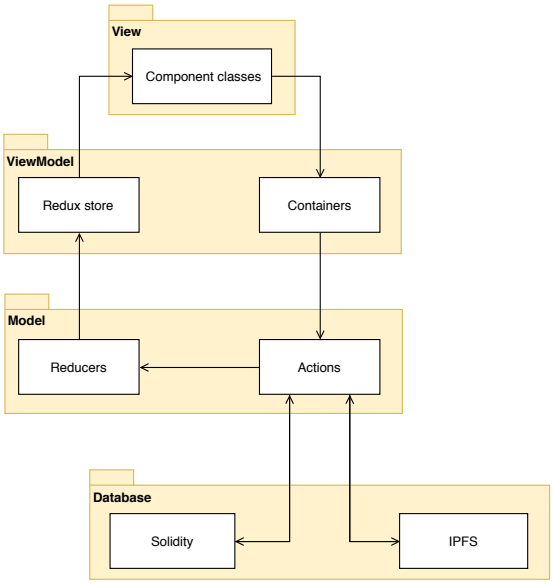


Figure 3: The general diagram of the packages

5 Esempio Javascript per giovanni

6 FrontEnd

This section of the manual regards the programming of some features that will make the user more comfortable with the package. By explaining how to do modify or add some functionalities, the reader will be guided into the logic of the application. Since the package has been developed using the *Model View View-Model* design pattern, it is recommended to follow those guide lines, as they will help to keep the package clean and efficient. In every section there will be a quick explanation of what the user is going to do and the steps that are needed for doing the tasks.

The guide will explain how to:

- Modify or add a *dumb* page whose functionality is to render some static information on the screen. This concerns the *View* section;
- Modify or add a page in which the user wants to render some `store` information by accessing its `state`, maybe paying attention to which kind of visitor the user wants to render the page to. This concerns the *View-Model* section as the information are already stored into the global `state` application;
- Modify or add a page in which the user can update the `store` information by making the user interacting both with interface and database. This concerns the *Model* section.

6.1 Modify or add a *dumb* page

We can call the *dumb* pages as `components`, as they all extends the `React` component abstract class. The only purpose of this kind of classes is to render static information on the screen or to make the user do some in-`component` tasks such as clicking a button to increase a counter.
To keep in mind:

- Each class is correlated by a `constructor` to which some `props` are passed by the parent `component`. It is possible to define a `state` with which the reader can manage the data; those are not meant to be used outside the component;
- The class should be correlated by a *SassCSS* style sheet to make it looks more personal. The file should stay along with the file of the component;
- All the components must stay into the `src/components/` folder under the respective field. So if the reader wants to add a component that will be rendered, for example:
  - along the *navigation bar*, the component should stay under the `App` folder;
  - if it is for some specific user, under `src/components/Profile/<userType>`;
  - if it is in common with all the users, under `src/Profile`;
  - if it is not user specific, under `src/components`.

  It is important to say that all the related files should be grouped together into the same sub folder. This is a Duck reference;
- To render a component we have to make the application be aware of it! So the `src/index.js` must be updated accordingly;

So let's say the reader wants to add a `BetterHome` homepage for the application. He will have to follow those steps:

1. Create a valid `React` component under `src/components/BetterHome/`;
2. Add a valid `React` button under `src/components/App` that must be imported into the `src/components/App/NavButto` we want to access the component we are going to create, aren't we?

3. Open `src/index.js`, import the `BetterHome` component;
4. Under `ReactDOM.render()` find the right place in which the component should stay and give it a `path` name. There are so many "..."`IsAuthenticated()`: we will discuss about it later. The button we have created before should be linked to that path.

What have we just done? We made the application to know about our new components and then render them when the user clicks on the navigation bar button. When the navigation button is clicked an history-`action` is triggered, the `state` is updated; by doing so the `ReactDOM.render()` method of `index.js` is updated and loads the page. Say thanks to the `React` framework!

6.2 Modify or add a *not-so-dumb* page

Adding a dumb component was ok, not so thrilling though. This chapter will introduce the reader into adding a `container`, which is responsible for *passing* a `props` to the previous defined `component` to make it more clever.
To keep in mind:

- The Marvin package owns a *Redux* `/src/store.js` in which the user can redefine the behavior of the application. In this file we compose some specific methods for managing the *history* of the application, the kind of `dispatch`es it will have to manage and the `action`s that the `reducer`s will have to trigger;
- Every `container` should stay under the `src/containers` sub folder following the rules we described above;
- The `container` is a special javascript class which is then *decorated* by a component. In this class we can *connect* a component to the `store` state so it has access to the information that are kept into it.

Let's say we want to make our `BetterHome` component renders some user's information, such as its own name. Then we have to follow those steps:

1. Just as creating the `component` `BetterHome`, create a `/src/containers/BetterHome/BetterHomeContainer.js` file accordingly to the rules we described above. In this class we will import the `BetterHome` component. Then, we will use the `connect` statement from `react-redux` to connect the `BetterHome` `state` to the `store` `state.user.data`; this data is now accessible from the components via `this.props.<dataNameInContainer>`;
2. After those steps, our `container` class is a *React* class, so it can be exported as an enriched component;
3. We should update the `BetterHome` component to make it renders the name of the user which is logged in;
4. We have to modify the import inside `src/index.js` to make the application know that we connected our component to the store. To achieve this we just switch import from the `component` to `container` class.

With those few steps we have just connected our *dumb* component to the `store` `state`. So every time the information will change, our `BetterHome` component will change accordingly.
Remember all those "..."`IsAuthenticated()` methods inside `src/index.js` component? Take a deeper look: instead of managing all the authorization, the SoS team choosed to use the `react-auth-wrapper` package. This tool is natively linked to the *Redux* store and decouples the authentication from the components. We created a `src/authentication/wrapper.js` file which can read the store information and create some `predicate`s in order to:

- Take a *React* component as parameter and decide to render it or not;
- Dispatch redirections in case of success/failure of the predicate.

For further explanations please refer to its GitHub page.

6.3 Modify or add a *very clever* page

Connecting a component to the store state is great, but we want something more from our application. Let's say that we want to add some other profile information such as the phone number. This is the very juicy part of the application - and the most rich of notions indeed. We will have to keep in mind some things:

- The data are stored into two different kind of database: the *blockchain* and the *IPFS* ones. We described what they are and which one is to be preferred for storing data in the *backend* part. The *IPFS* stores some information, it retrieves an *hash code* which is then written on the *blockchain*. We should decide to put our phone number into *IPFS*;
- As we are developing a `react-redux` application, we have to focus on what the reducer is: the reducer is a javascript file which is responsible to update the state information on the base of the actions that are dispatched. A user can dispatch an action: this will update the store state and trigger the application to re-render. Every reducer should stay under `/src/redux/reducers`;
- Even if there is no need to understand what *Web3* is as the settings should not be modifyed, some skilled developers would find interesting how to change the communication between the javascript part, the Metamask plugin, the *blockchain* network and the *solidity* contracts. *Truffle* uses this package to make a comfortable adapter to the *backend*, retrieving a *.json* instance of the *solidity* contract and making it available for interacting with javascript;
- As our application will make many asynchronous calls it is suggested to use *promises* instead of *callbacks* so this will keep the state coherent. If your application has to call a callback then you should transform it into a promise. Please refer to the example you can find into `api/utils/ipfsPromises.js` where the team has already transformed a callback call from `ipfs-mini` package into a promise.

Let's proceed.
First of all the developer must have a clear idea of what he wants to add to the application. As we are updating new user's phone number we will need to:

1. Read the information that are already present on the *blockchain* and *IPFS*, so we will need the application to be aware of the reading state;
2. Keep them temporarly to make some modifications;
3. Add the information into the *IPFS* network and retrieve the hash and making the application know about the adding state;
4. Update the hash information on the *blockchain*.

With those steps in mind the developer will have a clearer ideas of the following steps. Go down the Rabbit's Hole!

1. First of all the developer should modify the BetterHome page to make it accept a number insertion and a Save button, which will be connected to the store later;
2. Secondly he should become familiar with the reducers that are present into the `src/redux/reducers`. The team chose to make:
   - `userReducer` for managing the login, logout and signup of all the users;
   - `<userType>Reducer` for managing all the actions that a particular type of user shall do (i.e adding an Academic Year for administrators)
   - `ipfsReducer` to maintain the reading and writing on/from *IPFS*;
   - `web3Reducer` to make the application know about the web3 connection state.

   In fact the developer is suggested to use one of the actions that are present in those reducers as they are tested to keep the store state coherent. However he could also add its own or modify the existing paying a bit more attention not to break the application global state;
   To use the reducer in a more familiar way there are some standard dispatched that the developer can use effortless. They can be found at `src/redux/actions/standardDispatches/`;

3. The user should now write the function to update the phone number. As for the components and containers the developer is suggested to write his action inside a `/src/redux/action/` folder accordingly. There are many actions from which take inspiration: it's important to use the right dispatch to make the Marvin store know about how the readings and writing are going on;
4. Now we have to make the application aware of the action. To achieve this we have to modify the BetterHomeContainer accordingly using the `mapDispatchToProps` function offered by the `react-redux` framework;
5. The last thing to do is to make the Save button trigger the action. It will be enough to connect its event to the container function, which will be accessible from `this.props.<functionName>`.

We made it! We just upgraded the Marvin application with the possibility to update also the phone number.

This guide is not finished yet, indeed this is a beta: as the developer will see there are some hide-and-seek tricks that makes this package faster and more efficient.

7 Backend

7.1 Basics of system architecture

7.1.1 Data and logic separation

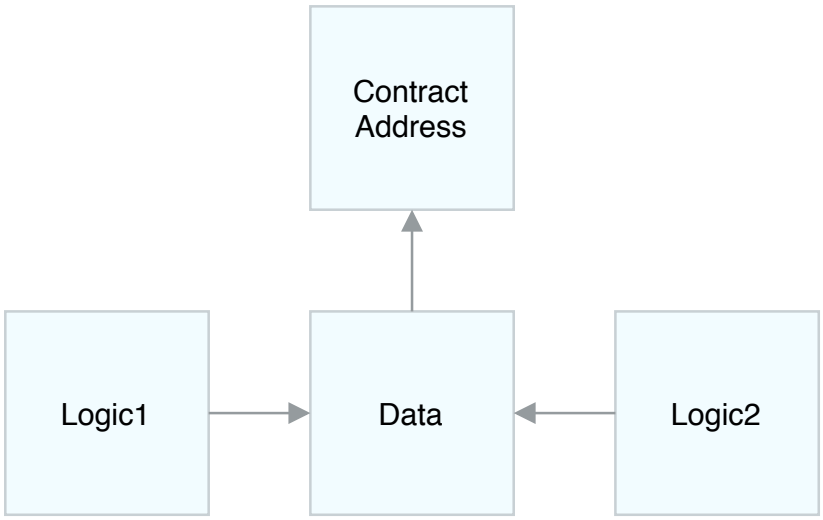


Figure 4: Data-Logic pattern

Due to contract limited potential of upgrade and maintenance after the deployment on blockchain, contracts have been splitted in two different parts, each of ones is saved in the corresponding contract. This way one contract contains the data structure of an object and the other one contains the logical complex methods that refers to data. This pattern allow the developer to update and deploy only the new logical contract, keeping the data contract persistent and not losing all the old data.
The part that structures the data contains only the data and the methods that let the user to retrieve (getters) or modify (setters) them. Not all the users of the blockchain should have the right to access the data: the contracts on the blockchain are accessible by everyone. To fix this problem, the majority of the methods contains in their header an access modifier that give access only to the authorized users.
The third element of this pattern is the ContractAddress: this contract contains only the newest version of the contracts addresses. This is useful for two main reasons:

1. Easier contract structure: it's ordinary that a data contract is used by multiple logic contracts like in the example above; in this case it has to save and set all the related logical contracts address needed for the modifiers. Instead, using the ContractAddress, developer have to save only the ContractAddress address, in the Data contract, that will include all the other contracts addresses;
2. Second one is logically derived from the point above: if a logic contract is updated and re-deployed (and consequently has a new contract address), the developer should have had to manually set the new address in all the n-contracts that contain the old one. Using the ContractAddress, instead, the developer have to set the new deployed contract address only in the ContractAddress and all the n-contracts will be updated.

Consequently this pattern make it easier for the developer to update the contracts and involves lower costs in the long term.

7.1.2 Contracts description

There is only one instance of a contract. To obtain multiple instance of an object there are struct inside contracts, this way you can gather consistent data together. Mapping are used for saving reference to struct object and arrays to iterate on that reference: in Solidity you can't obtain the keys of a mapping, so you need to save that keys in the array. There is a brief summary of the application's contracts:

- **ContractAddress**: like we said before, this contract contain all the addresses of the contracts and the address of the university;
- **UserData**: contains the application's users data, saved in a struct called User. This contract also contain the getter and setter methods for the data private fields;
- **UserLogic**: contains methods useful for not registered or not logged user. It uses UserData;
- **ExamData**: contains the university exams application data, saved in a struct called Exam. This contract also contain the getter and setter methods for the data private fields;
- **CourseData**: contains the university didactic activities application data, saved in a struct called Course. This contract also contain the getter and setter methods for the data private fields. It uses ExamData;
- **DegreeData**: contains the university degree courses application data, saved in a struct called Degree. This contract also contain the getter and setter methods for the data private fields;
- **StudentData**: contains the university students' application mapping, used for simplify queries, considering that most users will be students. This contract also contain the getter and setter methods for the data mapping;
- **Student**: contains methods useful for students. It uses UserData, ExamData and StudentData;
- **Teacher**: contains methods useful for professors. It uses UserData and ExamData;
- **Admin**: contains methods useful for admins and university. It uses UserData, ExamData, DegreeData.

7.2 IPFS

Saving data on the Ethereum blockchain is expensive and you can store only simple type data (like integer or string). To mitigate this problem external distributed database is used: we choose IPFS for it's easy of use and because it's in a pretty advanced version. For instructions on how to use IPFS you can see this section. The developer should follow this guidelines to choose to save an information on blockchain or IPFS:

1. Data used to query on contract should be saved on blockchain to prevent high latency due to backend-frontend and IPFS communication;
2. Data that could have concurrency access across different users should be saved on blockchain because Ethereum resolve concurrency access and modification by default, so it's easier for the developer to manage this problem;
3. All the other data (for example users name, exam description or complex and big-sized data such as pictures or pdf) should be saved on IPFS.

7.3 How to deploy contracts

Before deploying your contracts, you have to set migration javascript files. You can follow this online guide to know how to modify your migration file. After this, you have two options for the contract deployment:

1. Local private deployment using Ganache: you can follow the guide in this section for the deployment on Ganache blockchain;
2. Public deployment using Ropsten: you can follow this guide for the deployment on Ropsten blockchain using Infura node.

After the initial deployment, if you want to upgrade a contract (for example contract named A) and deploy the new one (named NewA), the new contract address have to be saved in ContractAddress contract; this way all the contracts that used A are updated with the NewA address and can allow it to access the data methods. It is not mandatory that the address that deploy the new contract is the university one, but only university address is allowed to set the new contract address in ContractAddress.