
**8Lab Solutions - Project "Soldino"**

# Developer manual

| | |
|---|---|
| **Version** | 0.1.0 |
| **Approval** | |
| **Drafting** | Mattia Bolzonella |
| | Samuele Giuliano Piazzetta |
| **Check** | Giacomo Greggio |
| **State** | Verified |
| **Use** | External |
| **Adressed to** | Red Babel |
| | 8Lab Solutions |
| | Prof. Tullio Vardanega |
| | Prof. Riccardo Cardin |

**Description**
Developer manual made by *8Labs Solutions* for the making of the project *Soldino*.

8labsolutions@gmail.com

# Changelog

| Version | Date | Name | Role | Description |
|---|---|---|---|---|
| 0.1.0 | 2019-04-08 | Giacomo Greggio | **Verifier** | Verification. |
| 0.0.7 | 2019-04-03 | Samuele Guliano Piazzetta | **Programmer** | Drafted §8. |
| 0.0.7 | 2019-04-02 | Samuele Guliano Piazzetta | **Programmer** | Drafted §6, §7. |
| 0.0.6 | 2019-03-30 | Mattia Bolzonella | **Programmer** | Drafted §5. |
| 0.0.5 | 2019-03-29 | Mattia Bolzonella | **Programmer** | Drafted §4. |
| 0.0.4 | 2019-03-29 | Samuele Guliano Piazzetta | **Programmer** | Drafted §3. |
| 0.0.3 | 2019-03-28 | Mattia Bolzonella | **Programmer** | Drafted §2.2, §2.3, §2.4. |
| 0.0.2 | 2019-03-28 | Samuele Guliano Piazzetta | **Programmer** | Drafted §1, §2.1. |
| 0.0.1 | 2019-03-27 | Samuele Guliano Piazzetta | **Programmer** | Created the structure of the document. |

# Contents

# List of Figures

# List of Tables

# Introduction

## Manual contents

This document is the developer manual of the project *Soldino*, developed by *8Lab Solutions* team for the proponent *Red Babel*.
Within the manual you can find:

- the technologies used for the development;
- the software tools used and suggested;
- the software architecture;
- the architectural and design pattern used;
- the functionalities provided by *Soldino*.

## Purpose of the manual

The contents of the manual are intended to help the developers who decide to maintain or further develop *Soldino*. Everything described here can help the developer to fully and deeply understand the design, use and features of the application, so that it can be modified and improved with ease.
Many technologies, tools and languages are used to build the app: these are only briefly explained in their parts that cover the application domain. Additional references can be found in the "Reference" section.

## Purpose of the product

*Soldino* platform is a DApp accessible on a web browser as a client interface and the plug-in Metamask is a virtual wallet used for transactions made through *Soldino*.
The main functionality of the product is trading goods and services online. Since the platform's backend is based on the Ethereum network, it provides more security and transparency than the traditional e-commerce websites.
The platform is built to be managed by the government and the currency used is called Cubit, it is a ERC20 compliant fork of Ether minted and managed by the government itself.

## References

- **Ganache** https://trufflframework.com/ganache
- **Git** https://it.atlassian.com/git
- **Node.js** https://nodejs.org/it/
- **Node Package Manager** https://www.npmjs.com/get-npm
- **Surge.sh** https://surge.sh/help/getting-started-with-surge
- **Truffle** https://truffleframework.com/truffle
- **Solidity** https://solidity.readthedocs.io/en/v0.5.6/

# Setup

## Requirements

In this section all of the requirements needed are described.

### Browser

*Soldino* is accessible through a web interface. The currently most recent versions of the following broswers are supported:

- **Mozilla Firefox**: version 64 or later;
- **Google Chrome** version 71 or later.

### Tools

The following tools are needed:

- **Git**: a famous control version system: *Soldino* is hosted on GitHub;
- **Node.js**: a framework needed for installing dependencies;
- **Truffle**: needed to write and deploy contracts with ease;
- **Ganache**: needed to put up a local Ethereum network and check transactions in it;
- **Metamask**: a browser plugin used as a virtual wallet;
- **Surge.sh**: a web platform chosen for hosting the website interface of Soldino.

### Dependencies

Soldino depends on many different packages, some for use and others for development.
All these packages are located in the file `package.json` which is in the root folder of the project.
The packages required to execute the software *Soldino* are listed below.

Table 2.1.1: Packages required for software usage

| Software | Version |
|---|---|
| react-text-mask | ≥5.4.4 |
| commondir | ≥1.0.1 |
| history | ≥4.7.2 |
| prop-types | ≥15.7.2 |
| react | ≥16.8.3 |
| react-dom | ≥16.8.3 |
| react-number-format | ≥4.0.6 |
| react-redux | ≥6.0.1 |

Table 2.1.1: Packages required for software usage

| Software | Version |
|---|---|
| react-router | ≥4.3.1 |
| react-router-dom | ≥4.3.1 |
| react-router-redux | ≥4.0.8 |
| react-scripts | ≥2.1.8 |
| redux | ≥4.0.1 |
| redux-thunk | ≥2.3.0 |
| web3 | 1.0.0-beta.37 |

Other packages, listed below, are required for the development.

Table 2.1.2: Packages required for development

| Software | Version |
|---|---|
| eslint | 5.12.0 |
| eslint-config-airbnb | ≥17.1.0 |
| eslint-loader | ≥2.1.2 |
| eslint-plugin-import | ≥2.16.0 |
| eslint-plugin-jsx-a11y | ≥6.2.1 |
| pre-commit | ≥1.2.2 |
| truffle-contract | ≥4.0.6 |

## Installing

### Browser

The first thing is to have your browser installed. You can get the latest chrome version here.

### Git

You should type this script in the shell for installing Git packet: `sudo apt install git`.

### Node

For installing Node.js you have to digit in the shell the following commands:

1. `curl -sL https://deb.nodesource.com/setup_11.x | sudo -E bash -`
2. `sudo apt install -y nodejs`

3. check that `node` have been installed correctly with `node -v`.

There is no need to install npm separately, since it is automatically installed with Node.

#### Truffle

You should check you have the truffle requirements:

- an OS among Linux, Windows and MacOS (prefer Linux);
- NodeJS v8.9.4 or later (we picked version 11);
- Node Package Manager (npm).

then you can install Truffle by running the command: `npm install -g truffle`

#### Ganache

There are three step to install Ganache:

1. you can download the Ganache executable at this link, clicking on the download button;
2. if you have a Linux operating system, you have to give the permissions to make the Ganache file executable. This can be done with the command
   `chmod +x path-of-the-appimage/name-of-downloaded-file.AppImage`;

#### MetaMask

You can add it to your browser in this way:

- Chrome: https://chrome.google.com/webstore/search/metamask?hl=it;
- Firefox: https://addons.mozilla.org/it/firefox/addon/ether-metamask/?src=search.

#### Surge

You have to install Surge by executing the shell command: `npm install -g surge`.

### Configuration

This section shows how to configure your work environment, so that it's the same as ours, in order to minimize the number and entity of troubles you will occur in.
Make sure to configure the tools in order. In particular, Truffle requires that Ganache is opened and is configured to execute successfully.

#### Cloning the repo

You have to clone the *Soldino* repository on GitHub: open the shell, move to the directory where you want Soldino to be, then use `git clone https://github.com/8LabSolutions/Soldino-PoC` and then use `git checkout final` .

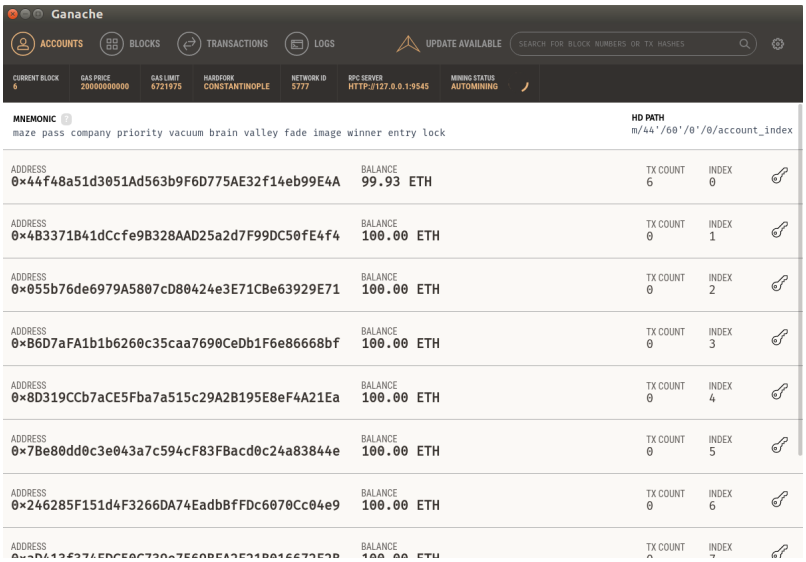


Figure 2.3.1: Ganache UI: from top to bottom you can see the menu bar, the current configuration, the mnemonic, and the interface of the selected menu option

### Ganache

You have to go to the folder where you've put Ganache, and open it with double click. Then you have to click on the cog at the top left corner to access Ganache settings and make sure you've matched the following settings (most of which are defined in `truffle-config.js`) on each respective window:

- Server:
  - hostname: `127.0.0.1`;
  - port number: `9545`;
  - network it: any (you can keep the default one).
- Account & keys:
  - nothing to configure here, but you should have a look at the **mnemonic**: it will help you later.

### Truffle

The configuration of Truffle is defined in the file `truffle-config.js` in the root directory of *Soldino*. On your shell type the following shell commands:

- `truffle console`
  opens the truffle environment in the shell under the configuration defined in truffle-config.js. From now on, every command is executed in the truffle environment, from which you can exit double typing `ctrl + C`.

- `compile`
  to compile the contracts: these are compiled in in a .json format (which enables interaction with the frontend) and put into the folder defined in `truffle-config.js` at `contracts_build_directory` (in our case the location is `./src/contracts_build`)
- `migrate --network development`
  puts the contracts runnning on the blockchain.

### MetaMask

It's necessary to create an account.

- open MetaMask on your browser
- select get started
- select import wallet
- use the seed frase (AKA the mnemonic) copy pasting the one
- from ganache and put your password

Now your account is up and synchronized with Ganache. Now you have to connect the wallet to Ganache network. Ganache settings are exposed in the Ganache UI just above the mnemonic. Let's synchronize MetaMask to the same network. On the top right corner there is a drop-down menu to select the network. Select `Custom RPC` to set your own local network.
You are now in the MetaMask advanced settings screen. In the field `Net Network` click on `Show Advanced Options` to open the form, then insert these data:

1. **New RPC URL**: http://127.0.0.1:9545 (the port number matches the one in truffle-config.js);
2. **Nickname**: the name you wanna give to your network;
3. **save** when you're done.

MetaMask is now connected to Ganache.

To enable transactions on your local test network, you can find free ether for testing networks on this site: https://faucet.metamask.io/.

## Running

Now that you have all the required software installed and configured, it's time to get it up running.
All you have to do is moving to the root directory of Soldino and prompt

```
npm run start
```

This will open the website on your browser at the address `http://127.0.0.1:3000` and allow you to explore it.

# React

## Overview

React is a JavaScript library, based on `npm` and made by Facebook, for building user interfaces by assembling user-defined components.

## Components

Components are the core of this `npm` module. *Soldino* is made of two types of component:

- Presentational components;
- Container components.

### Presentational

Each presentational component implements `Component` interface provided by the library. According to this interface, some methods are inherited from the presentational components: in particular our components uses `Render()` method to render themselves. Most components can be customized with different parameters when they are created. These creation parameters are called `props`. Each presentational component returns a single HTML tag, here we can customize the returned tag with some Bootstrap classes. The props are accessible by the components that own them, referencing them with `this.props.propsName`.

### Containers

This type of component is like a wrapper of presentational components. A function called `Connect(arg1, arg2)` is needed for connecting a container component to a presentational component: in this way we can map some actions and some application state variables inside the presentational component, passing them through props. The `arg1` is a function called `mapStateToProps()` and the `arg2` is another function called `mapDispatchToProps()`.

## React-Router

React-router-dom is a `npm` module used for rendering different pages without reloading the entire website, this module works with `Render()` method provided by each presentational component. *Soldino* is a single page application, so the router, implemented into a JavaScript file called `App.js`, is responsible to manage which components should be rendered.

## UML

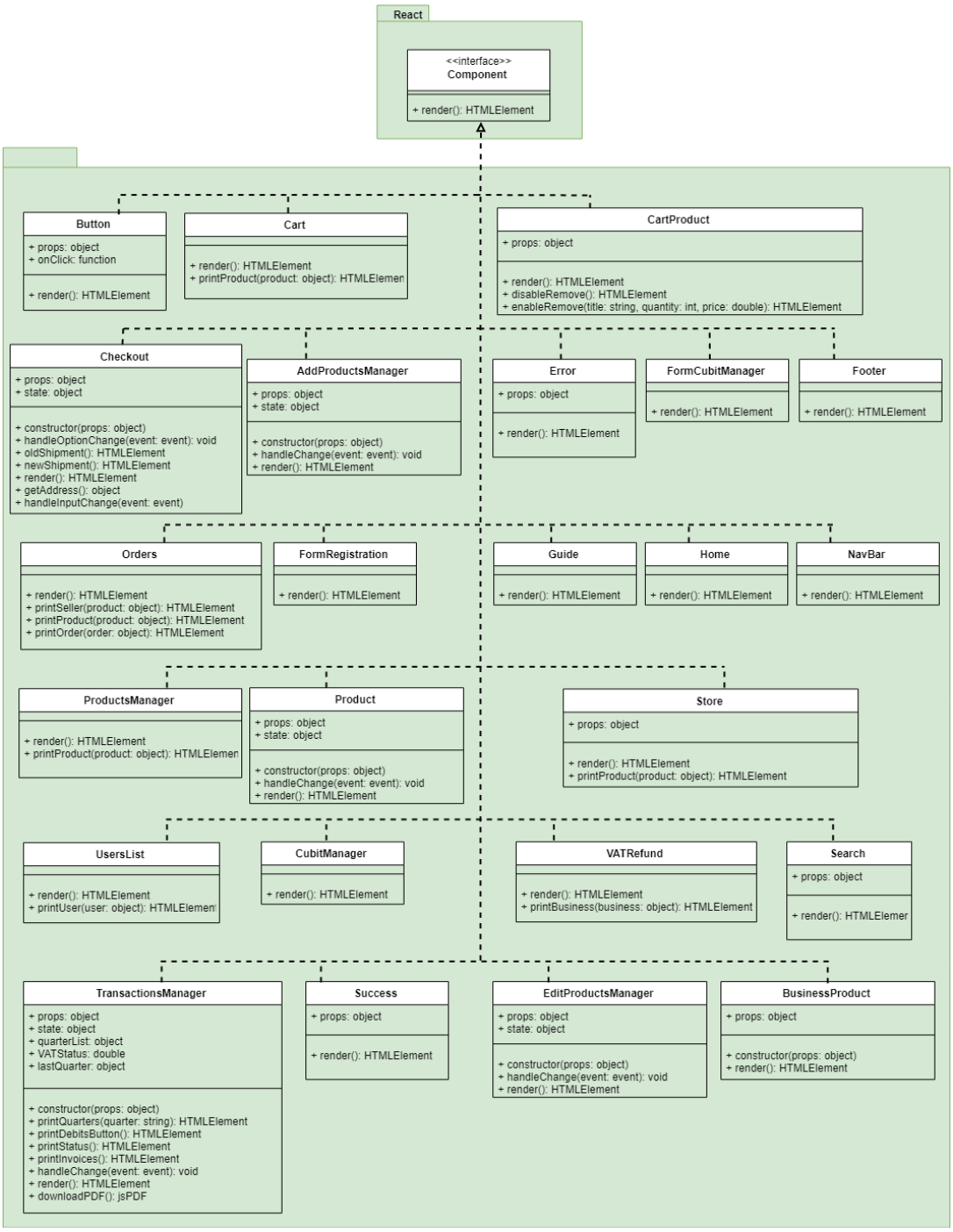


Figure 3.4.1: Presentational components

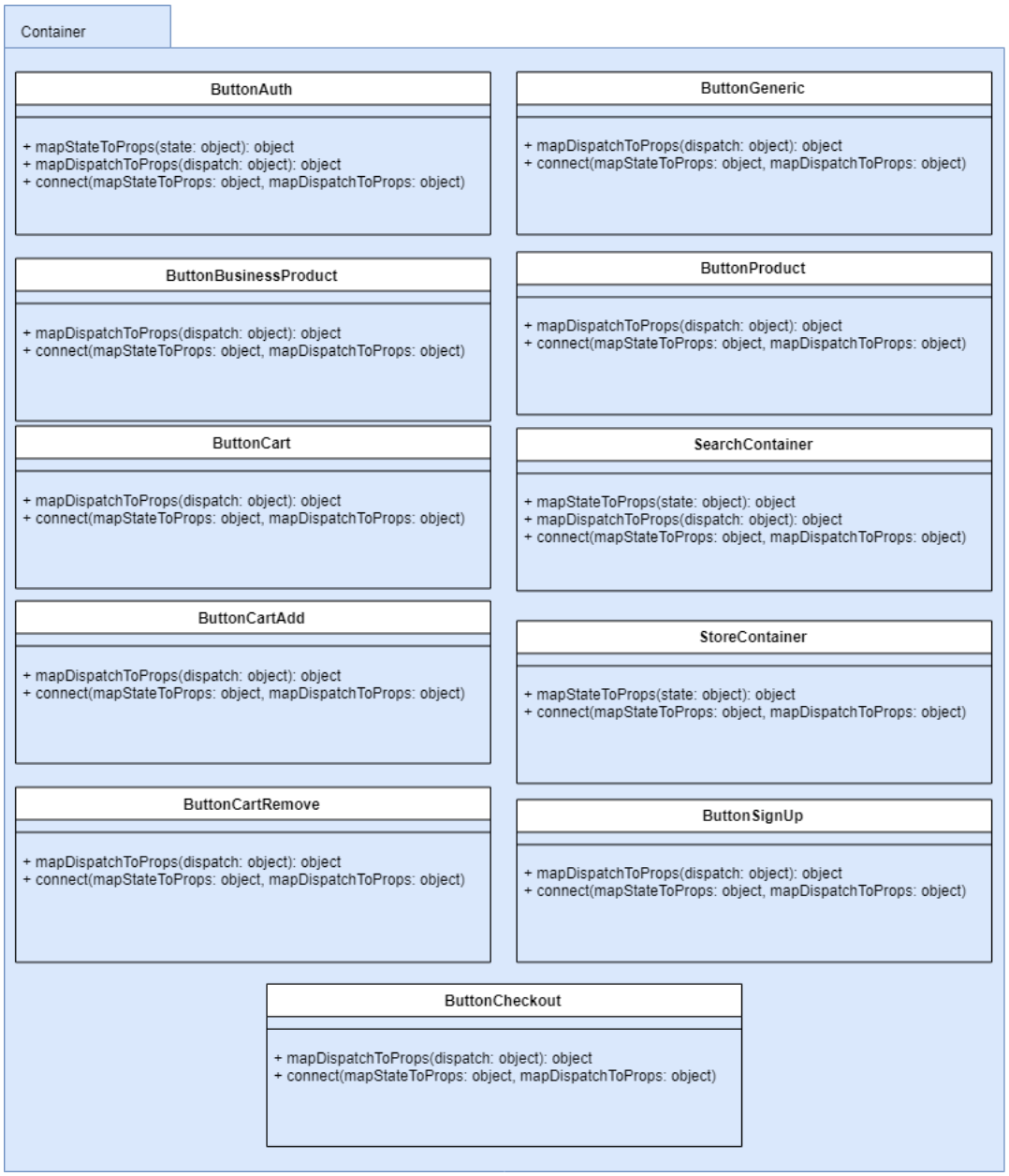


Figure 3.4.2: Container components

# Redux

## Overview

Redux is a `npm` module which manage the entire state of the website from client-side. It consists of:

- Actions;
- Reducers;
- Store.

When the website is built, a default state for the store is set (it is defined into the reducer JavaScript file).

## Unidirectional pattern

Basically some actions are mapped by `container components` into specific `presentational components` through them props with `Connect(arg1, arg2)` method. When a presentational component request a `dispatch()` of a specific action, a reducer will complete the request by changing the store and returning a new instance of the application state. Each time the store is changed, the `Render()` method of displayed components is called. `Redux` is the name of the pattern implemented by React and Redux, it is an evolution of `Flux` pattern, the difference is that Flux uses more than one store.

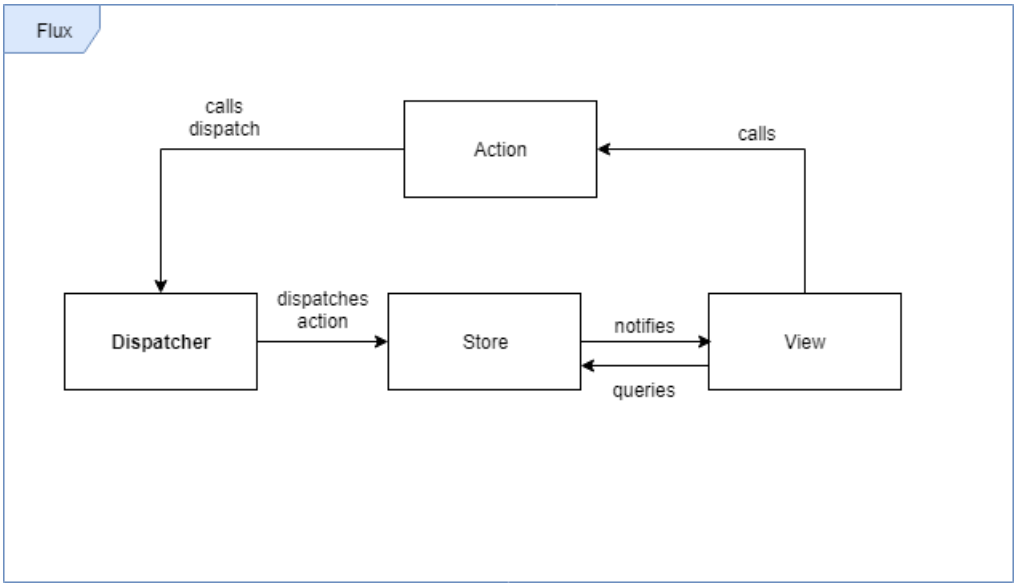


Figure 4.2.1: Flux pattern

## Actions

There are twelve different actions inside the `src/actions/` folder. Each action is called by the `action creator` when a container component request it's dispatch. Here there is the list of actions and what them do when they are invoked.

- **ACLogin**: it changes the `logged` state value;
- **ACIncreaseQuantity**: it increases the quantity of a product that is inside the cart, by changing the `quantity` value of a specified product inside `cart` state value;
- **ACCartToOrders**: it copies all the content of `cart` state value to `ordersList` state value, finally it resets the `cart` state value;
- **ACSignUp**: it provides to save into the store all of the user informations after they are successfully saved into IPFS;
- **ACAddToCart**: it puts the selected product with selected quantity into `cart` state value;
- **ACSearch**: it changes `searchProduct` state value for making a dynamic search inside store page;
- **ACLogout**: it changes the `logged` state value;
- **ACAddToSelling**: it puts the product into `selling` state value after it is successfully saved into IPFS;
- **ACReset**: it resets the entire state of application, even if `redux-persist` is present;
- **ACRemoveFromCart**: it removes the selected product from `cart` state value;
- **ACDecreaseQuantity**: it decreases the quantity of a product that is inside the cart, by changing the `quantity` value of a specified product inside `cart` state value;
- **ACDisableAccount**: it sets a value different from 0 to `error` state value if something goes wrong, otherwise it sets 0 into `error` state value.

## Reducers

- **ReducerAuth**: it makes the dispatch of the actions inside:
  - ACLogin;
  - ACLogout;
  - ACSignUp;
  - ACReset.
- **ReducerSearch**: it makes the dispatch of search action;
- **ReducerCart**: it makes the dispatch of the actions inside:
  - ACIncreaseQuantity;
  - ACDecreaseQuantity;
  - ACCartToOrders;
  - ACAddToCart;

  - ACAddToSelling;
  - ACRemoveFromCart.

- **ReducerGovernment**: it makes the dispatch of disableAccount action.

### Store

It's configuration resides into root reducer file, the initial configuration is:

- logged: false;
- user: null;
- searchProduct: "";
- cart: [ ];
- pending: [ ];
- ordersList: [ ];
- error: 0.

Each time the `reset` action is dispatched from it's reducer, the initial configuration of the store is set.

### Redux-Persist

It is a `npm` module used for maintaining the current store even if the user leaves the website. It is browser-locally saved, so if a user will enter into the website from another device or from another browser, the store will be set with default values. It has a blacklist for bypassing some values, in this way reloading the website, these values will not be saved (the blacklist is defined into reducer JavaScript file).

## UML

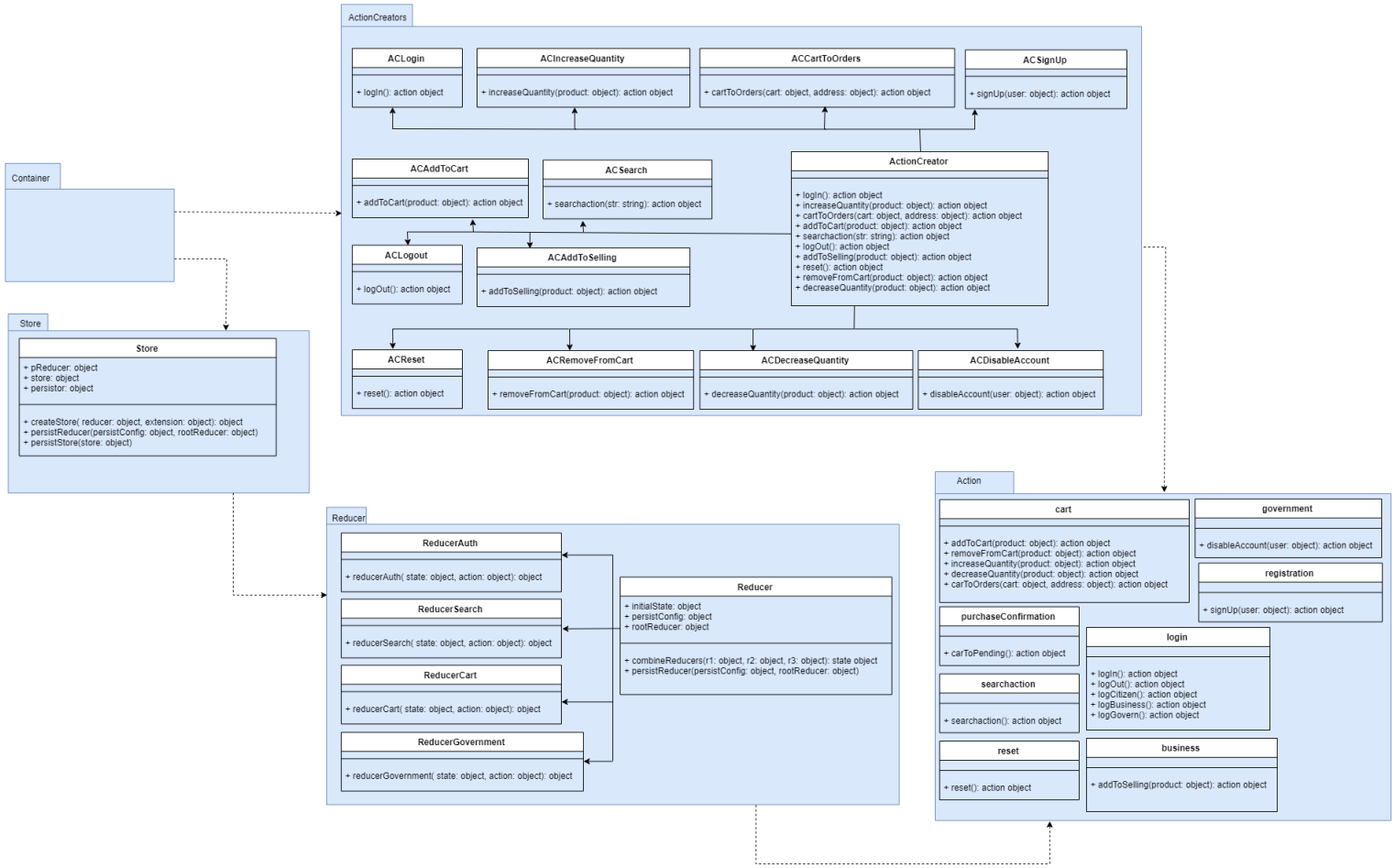


Figure 4.7.1: Redux architecture

# Web3

web3.js is a collection of libraries that allows you to interact with a local or remote Ethereum node. Moreover, it is able to communicate with MetaMask, which is the add-on for Chrome and Mozilla Firefox that lets the user manage his Ethereum wallet and confirm transactions in an intuitive and secure way. Using MetaMask we offer secure payments and automatic login to *Soldino*.

## Architecture overview

To interact with web3 library calls we organized the `web3functions` package into five different modules:

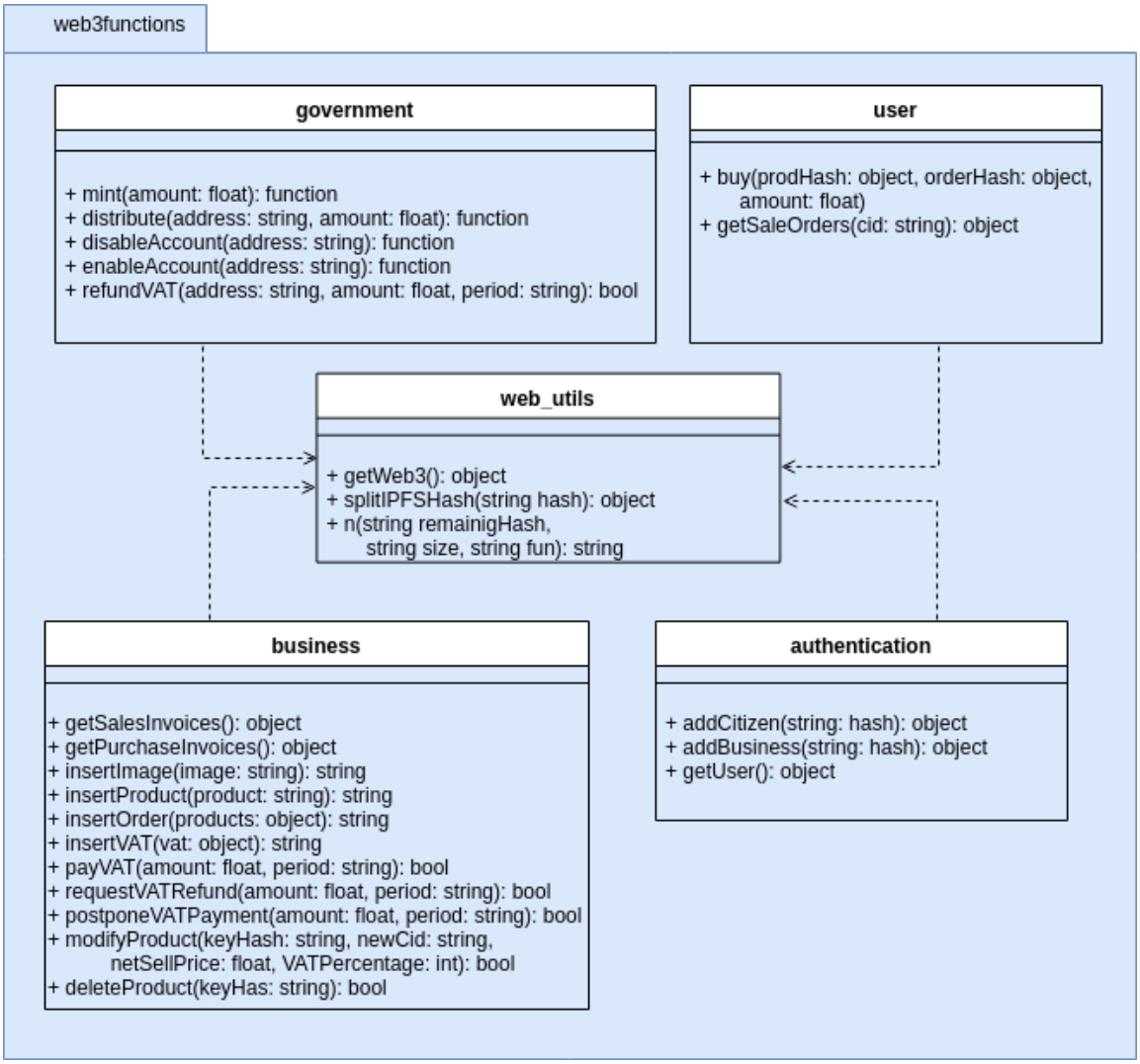


Figure 5.1.1: Class diagram of the `web3functions` package

## Methods

The five modules groups some methods that are responsible for setting data to Ethereum or retrieving them from the blockchain.

### web_utils

This module groups some utility functions that are used by the other modules to make the web3 call easier:

- **getWeb3**: tries to get an instance of a web3 object, which will be used to make all the web3 calls. Specifically, it tries to get the web3 object instance injected by MetaMask, if this is possible, otherwise it returns an error;
- **splitIPFSHash**: a function that converts the base58 string representing the IPFS CID into three variables, since this information is saved in this way in the blockchain for cost and scalability reasons;
- **recomposeIPFSHash**: it's the the inverse of the function descripted above, it recomposes the IPFS CID.

### authentication

This module manages the registration and login of all the types of users:

- **addCitizen**: registers a new citizen using the Ethereum address provided by MetaMask, and saves the IPFS CID, that represents the related information saved in the peer-to-peer network;
- **addBusiness**: registers a new business using the Ethereum address provided by MetaMask, and saves the IPFS CID, that represents the related information saved in the peer-to-peer network;
- **getUser**: returns the IPFS CID corresponding to the current Ethereum address of the user provided by MetaMask.

### user

This module manages the common functionality offered to citizen and business:

- **buy**: manage a new order making the user transfer the due amount of *Cubit* to the target companies;
- **getSalesOrders**: gets the IPFS CID related to an order. With the IPFS CID all the order's information can be retreived.

### business

This module manages the functionality offered to business:

- **getSalesInvoices**: returns an array containing all the IPFS CID related to the sales invoices;
- **getPurchaseInvoices**: returns an array containing all the IPFS CID related to the purchases invoices;

- **insertProduct**: inserts the product passed as JSON in the Ethereum blockchain;
- **insertOrder**: creates a new order made of the product passed to the function;
- **insertVAT**: inserts the VAT information related to the order passed to the function;
- **payVAT**: allows a business to pay the VAT owed by the government related to a particular VAT quarter;
- **postponeVATPayment**: allows a business to postpone the VAT payment to the government related to a particular VAT quarter;
- **modifyProduct**: allows a business to modify one of its product that was previously been added;
- **deleteProduct**: allows a business to delete one of its products that were previously been added. The product will not be shown in the store anymore.

**government**

Module to manage the common functionality offered to the government:

- **mint**: lets the government mint a specific amount of *Cubit*. This amount is deposited in the government account;
- **distribute**: lets the government transfer a specific amount of *Cubit* from his own account to a target account;
- **disableAccount**: lets the government set the state of the target account to "disabled". The target account will not be able to use the platform until it has been restored;
- **modifyProduct**: lets the government set the state of the target account to "enabled". The target account will be able to use the platform;
- **refundVAT**: lets the government refund a business. The owed amount of *Cubit*, related to a specific VAT quarter, will be transferred to the target business.

# IPFS

It is well known that the Ethereum blockchain is not suited for saving data. You just need to know that for hypothetically uploading a file on Ethereum, you would have to pay about 2$ per kilobyte.
For these reasons, we adopted the InterPlanetary File System (IPFS) protocol in the homonymous peer-to-peer network to store all the information that is not involved in VAT management. Specifically, for security purposes, all the data used to make payments between the clients are exclusively stored in the Ethereum blockchain. All the remaining data, such as not sensitive clients' data, products' data, orders' data, are stored into IPFS since it is a free and distributed technology born with the aim of sharing every kind of content.

## Architecture overview

To establish a connection to the IPFS network we developed a small package that is responsible for hiding the IPFS connection implementation (`ipfs-mini` API). The available interface contains two methods: one for adding new data into the blockchain, and the other one for retrieving them starting from the IPFS content identifier (CID).

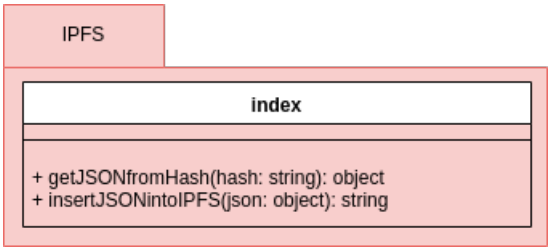


Figure 6.1.1: Class diagram of the IPFS package

## Methods

The package provides the following methods:

- **getJSONfromHash**: it uses the `ipfs-mini` API to retreive the JSON object corresponding to the passed IPFS CID;
- **insertJSONintoIPFS**: it uses the `ipfs-mini` API to upload the JSON passed to the function, then returns the IPFS CID.

# Facade

## Architecture overview

Since most of the operations include the interaction with both web3funcions and IPFS packages, we provide a third package, facade, with the aim of hiding the web3 and IPFS layers. This package executes function calls in the suitable order to the other two packages, saving and retrieving data from both IPFS and Ethereum.

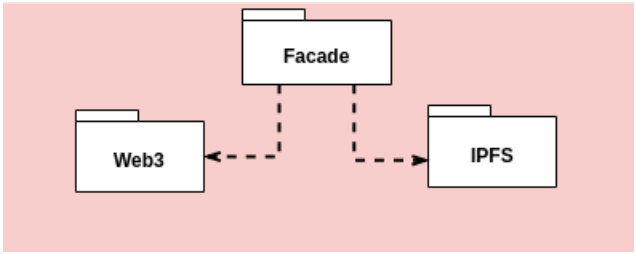


Figure 7.1.1: Package diagram with the interaction between facade-web3-IPFS

## Methods

All the methods described in the picture below consist of two different steps:

- **setter**: first the data is stored to IPFS, then it is stored to Ethereum with the related IPFS CID;
- **getter**: the IPFS CID is retrived from web3. The CID is then used to get the the data from IPFS.

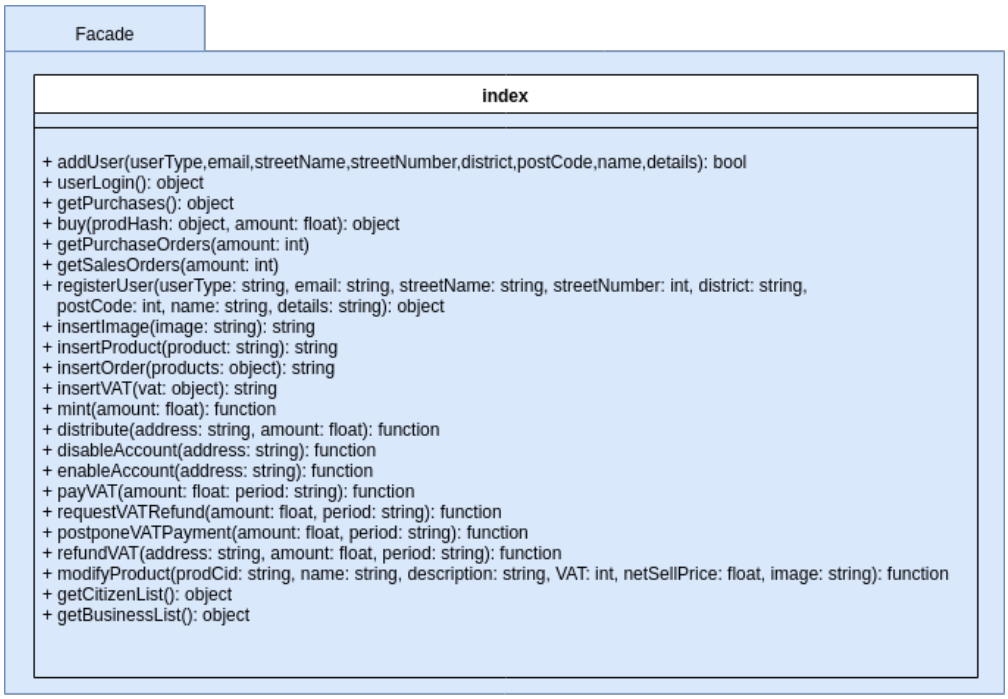


Figure 7.2.1: Class diagram of the facade package

Since the methods are just a combination of the already described methods of the web3 and IPFS packages, we omit their description.

# Solidity

## Architecture overview

As you know the main feature of the blockchain is its immutability. It means that once a smart contract (referred as contract from now on) is deployed, it's not possible to modify or update it. In order to update a contract you need to deploy a new contract on the blockchain.
In view of this fact and the proponent's request to develop upgradeable smart contracts, the architecture ideated and developed reflects these two aspects. As a matter of fact smart contracts in Soldino are essentially of three types:

- **Storage contract**: this type of contract isn't upgradeable because it's used to save all the critical data like products, orders, VAT movements;
- **Logic contract**: this type of contract implements the business logic. In addition a logic contract as a reference to its storage contract (e.g. UserLogic has a reference to UserStorage). This choice has been made to improve the security of the storage contract (it will be explained in section 7.2.2);
- **Generic contract**: this type defines the contracts that aren't either logic nor storage. They are:
  - **TokenCubit**: contract to implement the custom token ERC20 compliant *Cubit*;
  - **ContractManager**: contract used to achieve simple upgradeablilty of the system;
  - **Purchase**: contract to allows the user to buy several products with minimum number of transaction;
  - **Owned** and **Authorizable**: contracts used for the system's security;
  - **Migration**: contract provided by Truffle framework, it's used to deploy all contracts.

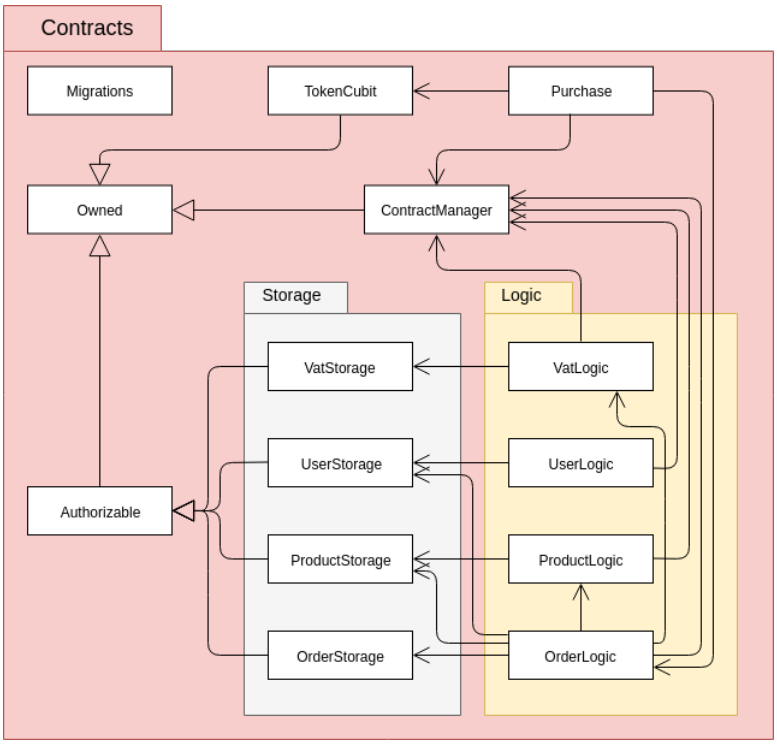


Figure 8.1.1: Simplified class diagram of the contracts (state variables and methods are omitted)

## Contracts

*Disclamer: the class diagram for each contract may be updated in the future to better represent the final product. Updates concern cost or performance optimization.*

### Generic contracts

#### 8.2.1.1 Security contracts

**Owned** This contract defines the owner of a contract. In Soldino the owner is the address which deploy all contracts. The contract defines a modifier, `onlyOwner`, used to allow only the owner to call a function which uses the above-mentioned modifier.

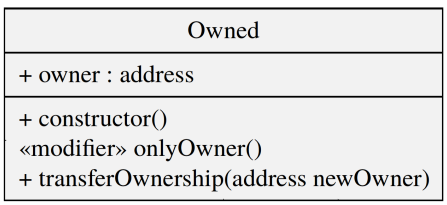


Figure 8.2.1: class diagram of the Owned contract

**Authorizable** This contract is used by all storage contracts. It's a derivate contract of `Owned` and stores all the addresses authorized to call the functions that use the modifier `onlyAuthorized` defined in this contract. To authorize a contract the owner needs to call the function `addAuthorized`.

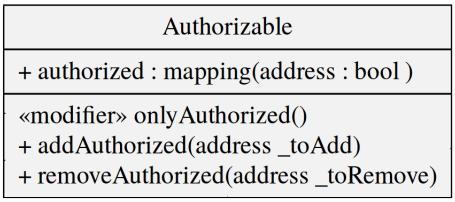


Figure 8.2.2: class diagram of the Authorizable contract

#### 8.2.1.2 Token ERC20

The contract `TokenCubit` implements the custom token *Cubit*, stores all the users' balances and defines all the methods in order to be ERC20 compliant.

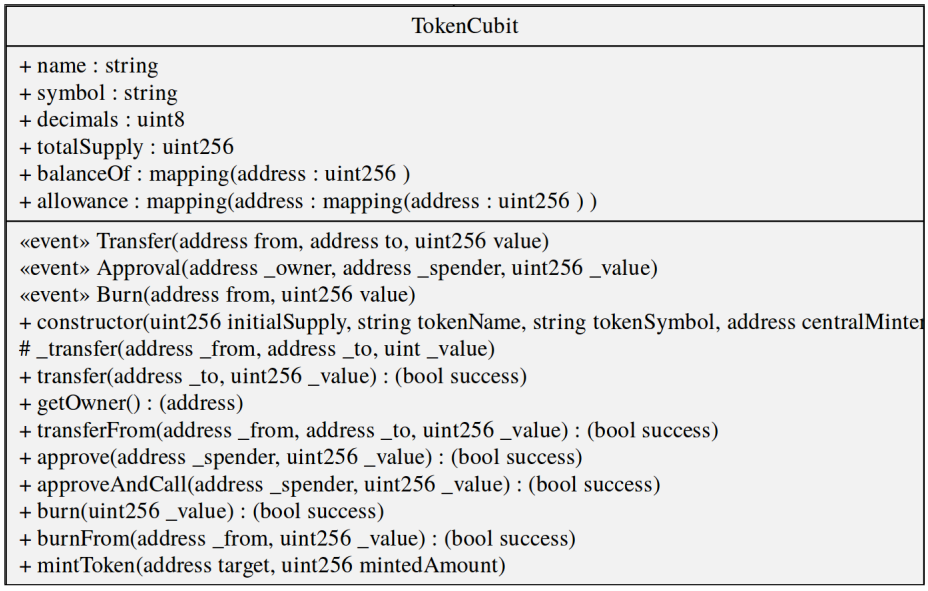


Figure 8.2.3: class diagram of the TokenCubit contract

#### 8.2.1.3 ContractManager

The purpose of the `ContractManager` is to achieve simple and cost effective upgradeability of the logic contracts. This contract stores a map in which the entries are composed in the following way:

- **Key**: the contract name;
- **Value**: the last version of the contract deployment address.

When contract $A$ needs to communicate with another contract $B$, contract $A$ get the address of the last version of the contract $B$ from `ContractManager`. In this way there's no need to manage all the references in the contracts.

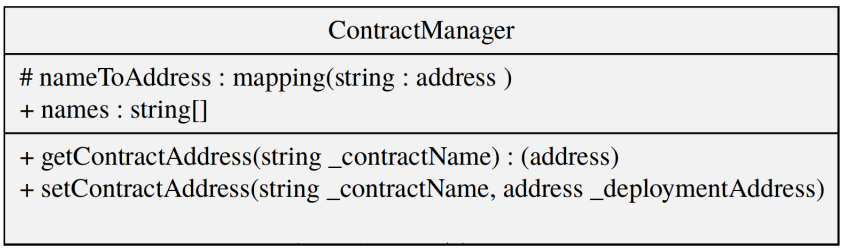


Figure 8.2.4: class diagram of the ContractManager contract

#### 8.2.1.4 Purchase

The `Purchase` contract acts as a façade when it comes to buy products on Soldino. Usually the user needs to confirm $n$ transactions for each order (intended as one order for each seller) to buy all the products in his cart, where $n$ is the number of orders.
With the `Purchase` contract, the user has to confirm always two transactions, no matter how many products are in his cart.

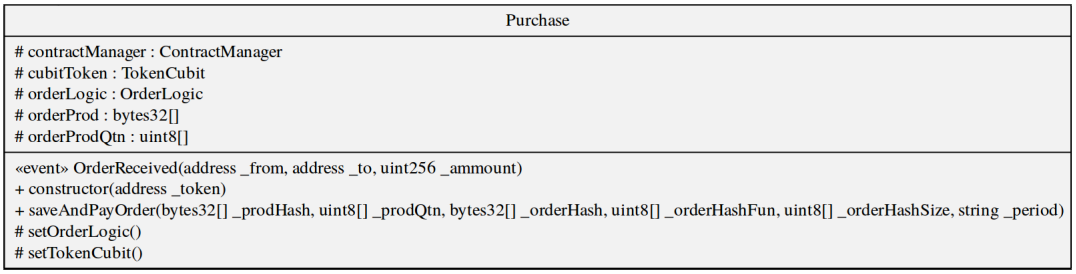


Figure 8.2.5: class diagram of the Purchase contract

**Storage contracts**

In this section we will illustrate how the storage contracts work. As mentioned before the storage contracts are immutable because they store all the critical data. When a contract is upgraded a new version of itself is deployed using inheritance, and all its state variables are new. To avoid data loss, all the data stored in the previous version of the contract should be copied into the new version. This copy is very expensive, due to high costs of writing on storage variables.
To avoid that, storage contracts implement don't implement any kind of business logic. Their purpose is to store data and to allow specific contracts to modify their state. For data retrieval, on the other hand, the are no limitations because getter methods don't modify the state of storage contracts. All storage contracts inherit from `Authorizable` in order to use the `onlyAuthorized` modifier in the setter methods.

**Note: struct in the storage contracts** All contracts, except for `VatStorage`, in their struct have three state variables: `hashIpfs, hashFunction` and `hashSize`, which represent the `IPFS CID` used to locate additional data on the `IPFS` network such as name and surname for users. The choice to separate critical data from additional data has been made in order to optimize storage costs.

#### 8.2.2.1 UserStorage

This contract maintains the VAT amount of every business for every quarter. Each time a business buys or sells some products, the VAT amount of each product is added or subtracted to the amount in this contract. If the product is sold, its VAT is added, whereas if the product is sold the VAT is subtracted.

This contract stores all the critical data of the users.

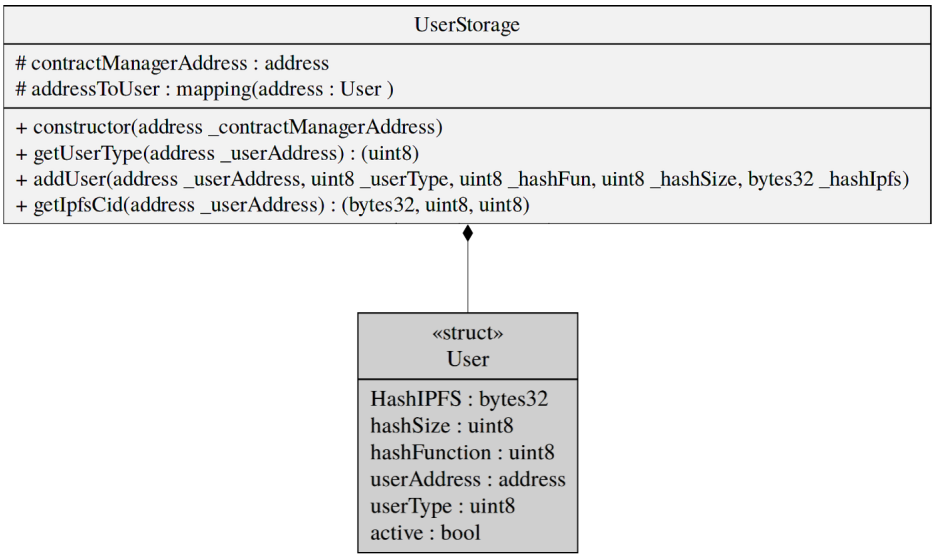


Figure 8.2.6: class diagram of the UserStorage contract

#### 8.2.2.2 ProductStorage

The `ProductStorage` contract stores all the data needed to be secured (e.g. net price). Furthermore, the contract defines all the methods used to maintain the data.

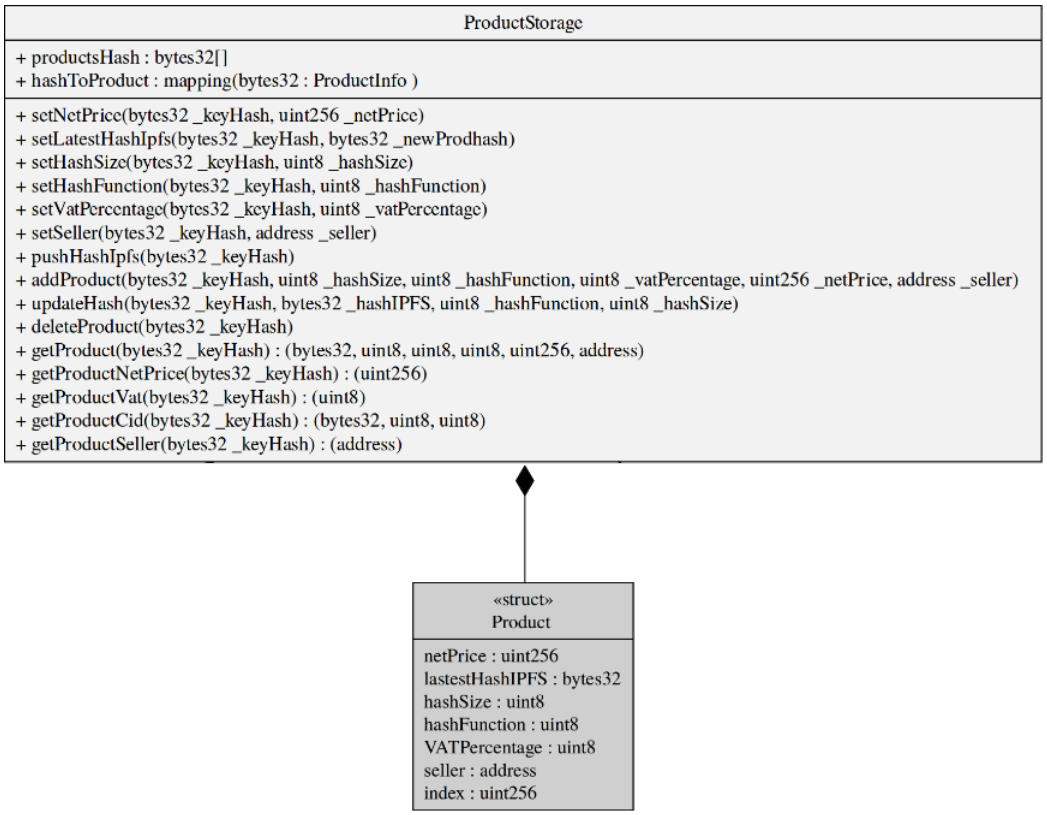


Figure 8.2.7: class diagram of the ProductStorage contract

#### 8.2.2.3 VatStorage

This contract maintains the VAT amount of every business for every quarter. Each time a business buys or sell some products, the VAT amount of each product is added or subtracted to the amount in this contract. If the product is sold, its VAT is added, while if the product is sold the VAT is subtracted.
`VatStorage` has an array of map keys in order to iterate the map and enumerate its elements.

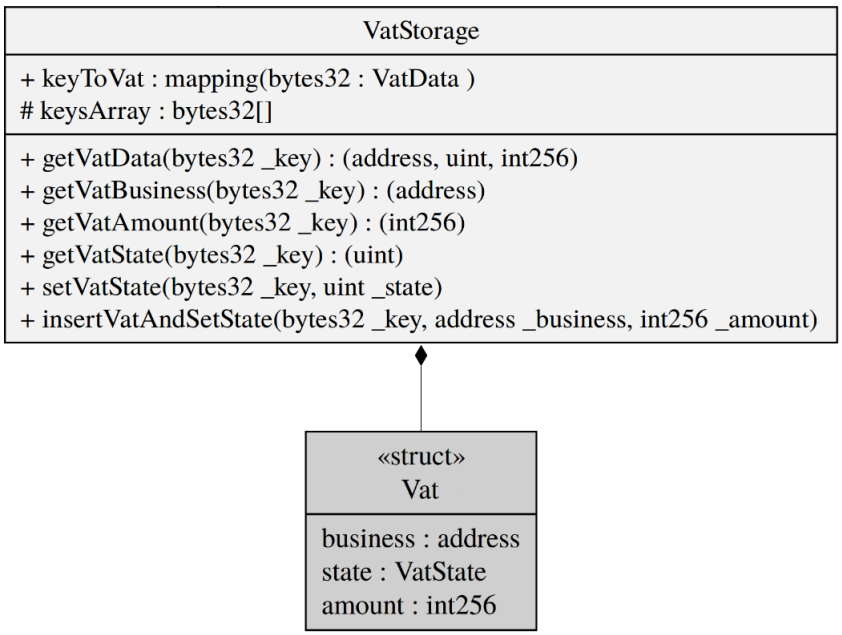


Figure 8.2.8: class diagram of the VatStorage contract

#### 8.2.2.4 OrderStorage

This contract stores all the information of an order. In its struct are saved also the product keys (`productsHash`) of the order.

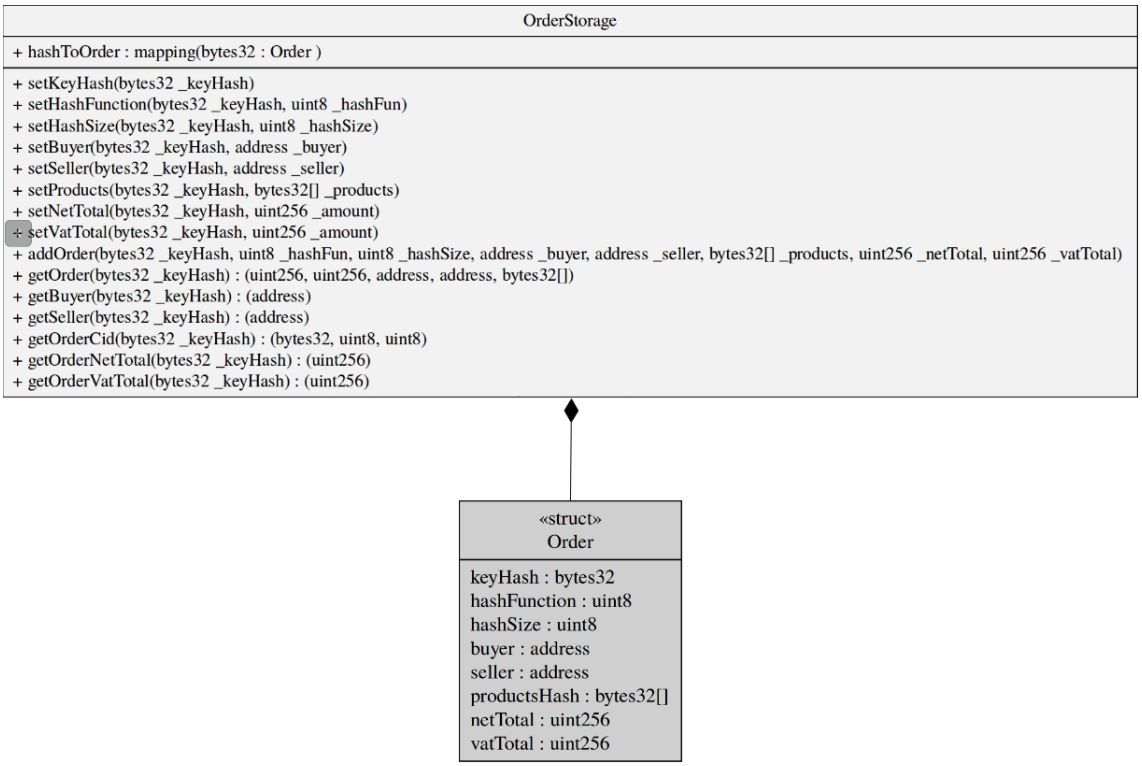


Figure 8.2.9: class diagram of the OrderStorage contract

**Logic contracts**

In this section we show how the logic contracts of *Soldino* work. These contracts implement a major part of the business logic, defining:

- inputs validation mechanics;
- events to be emitted on the blockchain when a state change is made;
- communication with each other.

Since logic contracts are upgradeable, every logic contract has a `ContractManager` state variable, used to get the latest address of another contract. In this way the coupling between logic contract is minimum and there's no need to maintain correct references in each logic contract.
However each `xLogic` contract has a direct reference to its `xStorage` contract because the latter is immutable.

#### 8.2.3.1 UserLogic

This contract provides an interface to comunicate with the `UserStorage` contract. `addCitizen` and `addBusiness` functions add a new user and set its type depending on which function is called.

| UserLogic |
| --- |
| # contractManager : ContractManager<br># userStorage : UserStorage |
| «modifier» onlyGovernment()<br>+ constructor(address _contractManagerAddress)<br>+ addCitizen(bytes32 _hashIpfs, uint8 _hashSize, uint8 _hashFun)<br>+ addBusiness(bytes32 _hashIpfs, uint8 _hashSize, uint8 _hashFun)<br>+ isRegistered(address _userAddress) : (bool)<br>+ login(address _userAddress) : (uint8)<br>+ getUserInfo(address _userAddress) : (bytes32, uint8, uint8)<br>+ setUserState(address _userAddress) |

Figure 8.2.10: class diagram of the UserLogic contract

#### 8.2.3.2 ProductLogic

This contract allows business to insert new products, modify or delete exiting ones. Obviously, only businesses can insert products in *Soldino*: to comply this requirement `ProductLogic` defines the modifier `onlyBusiness` which checks if the address of the contract that is trying to insert a product is a business. Furthermore, products can be modified and deleted only by their sellers and the modifier `onlyProductOwner` is responsible for that.

| ProductLogic |
| --- |
| # productStorage : ProductStorage<br># contractManager : ContractManager |
| «event» ProductInserted(bytes32 _keyHash, address _seller)<br>«event» ProductModified(bytes32 _keyHash, address _seller, bytes32 _newHashIPFS)<br>«event» ProductDeleted(bytes32 _keyHash, address _seller)<br>«modifier» onlyBusiness()<br>«modifier» onlyProductOwner(bytes32 _keyHash)<br>«modifier» onlyValidKeyHash(bytes32 _keyHash)<br>«modifier» onlyValidIpfsCid(bytes32 _hashIpfs, uint8 _hashFun, uint8 _hashSize)<br>+ constructor(address _contractManager)<br>+ addProduct(bytes32 _hashIPFS, uint8 _hashSize, uint8 _hashFunction, uint8 _vatPercentage, uint256 _netPrice)<br>+ modifyProduct(bytes32 _keyHash, bytes32 _hashIPFS, uint8 _hashSize, uint8 _hashFunction, uint8 _vatPercentage, uint256 _netPrice)<br>+ deleteProduct(bytes32 _keyHash)<br>+ calculateProductVat(bytes32 _keyHash) : (uint256)<br>+ calculateProductGrossPrice(bytes32 _keyHash) : (uint256) |

Figure 8.2.11: class diagram of the ProductLogic contract

#### 8.2.3.3 VatLogic

This contract defines the logic to interact with the `VatStorage` contract. Its function `refundVat` makes sure that only the government can refund the VAT to a business, thanks to the attached modifier `onlyGovernment`.
In order to store VAT movements broken down by business and quarter, the function `createVatKey` computes a valid key for the map stored in `VatStorage` using the address of the business and the string representing the quarter in format `"year-quarterNumber"`.

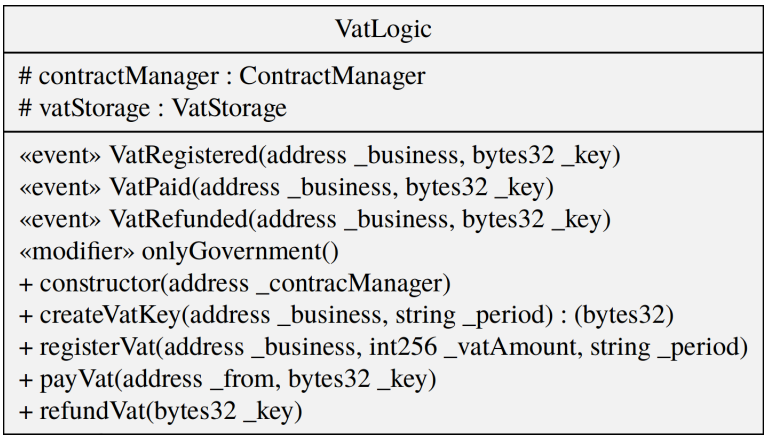


Figure 8.2.12: class diagram of the VatLogic contract

#### 8.2.3.4 OrderLogic

This contract allows to register an order. The function `registerOrder` can be called only by the `Purchase` contract because when a purchase is made on *Soldino* the latter contract is called and forwards the request to this contract. In other words, if a purchase contains products from different sellers, the `Purchase` contract calls `registerOrder` for every different seller.
In addition, registerOrder registers the VAT movements related to the businesses involved in the purchase.

| OrderLogic |
| --- |
| # contractManager : ContractManager<br># orderStorage : OrderStorage<br># vatLogic : VatLogic<br># userStorage : UserStorage<br># productStorage : ProductStorage<br># productLogic : ProductLogic |
| «event» PurchaseOrderInserted(address _buyer, bytes32 _keyHash)<br>«event» SellOrderInserted(address _seller, bytes32 _hashIpfs)<br>«modifier» onlyBuyerOrSeller(bytes32 _keyHash, address _user)<br>«modifier» onlyPurchaseContract()<br>«modifier» onlyValidIpfsCid(bytes32 _hashIpfs, uint8 _hashFun, uint8 _hashSize)<br>+ constructor(address _contractManagerAddress)<br>+ registerOrder(bytes32 _hashIpfs, uint8 _hashFun, uint8 _hashSize, address _buyer, string _period, bytes32[] _productsHash, uint8[] _productsQtn)<br># setProductStorage()<br># setVatLogic()<br># setProductLogic()<br># setUserStorage()<br># calculateOrderTotal(bytes32[] _productsHash, uint8[] _prodQtn) : (uint256, uint256)<br>+ getOrderTotal(bytes32 _orderHash) : (uint256)<br>+ getOrderSeller(bytes32 _orderHash) : (address) |

Figure 8.2.13: class diagram of the OrderLogic contract

## How to extend

As described above, smart contracts have been developed with the aim of being upgradable. This means that all the logic contracts can be upgraded without losing the stored data. In order to upgrade a logic contract you have to deploy the new contract, and update the contract manager, changing the address associated with the upgraded contract:

```
1 //import the new version of the contract
2 var Contract_v2 = artifacts.require("Name_of_the_contract");
3 //deploy the new contract
4 deployer.deploy(Contract_v2, contractManagerInstance.address)
5 .then((newContractInstance)=>{
6   //change the deployment address of the upgraded contract
7   //in the contract manager
8   return contractManagerInstance.setContractAddress("Name_of_the_contract",
        newContractInstance.address);
9 });
```