# Developer Manual

**SWEight Group - Project Colletta**

SWEightGroup@gmail.com

**Informazioni sul documento**

| | |
|---|---|
| **Version** | 1.0.0 |
| **Approver** | Damien Ciagola |
| **Writers** | Francesco Corti<br>Sebastiano Caccaro<br>Alberto Bacco<br>Enrico Muraro<br>Francesco Magarotto |
| **Verifiers** | Damien Ciagola<br>Georghe Isachi |
| **Use** | External |
| **Distribution** | MIVOQ<br>Prof. Vardanega Tullio<br>Prof. Cardin Riccardo<br>SWEight Group |

**Description**

The document contains the technical details a developer needs to comprehend and expand the software product

# Change log

| Version | Date | Description | Author | Role |
|---|---|---|---|---|
| 0.2.0 | 2018-04-01 | §4 completed | Sebastiano Caccaro | *Writer* |
| 0.1.1 | 2018-04-01 | Written §4 to §4.2.1 | Sebastiano Caccaro | *Writer* |
| 0.1.0 | 2018-04-01 | Backbone translated in English | Sebastiano Caccaro | *Writer* |
| 0.0.1 | 2018-03-18 | Document backbone | Enrico Muraro | *Writer* |

# Contents

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Document goal

## 1.2 Product goal

## 1.3 Glossary

## 1.4 References

### 1.4.1 Normative references

### 1.4.2 Informative references

# 2 Development Requirements

## 2.1 System requirements

### 2.1.1 Windows

- **CPU**: Intel X86 family;
- **RAM**: at least 2GB of RAM;
- **Disk's space**: at least 1GB;
- **Operating system**: Windows 7 or superior, 32-bit or 64-bit versions;
- **Java**: Java SE Development Kit 8;
- **Node.js**: Node.js 10.15.1;
- **Maven**: Maven 3.6.0;
- **Browser**: Any browser which support Javascript, HTML5 and CSS3.

### 2.1.2 Ubuntu

- **CPU**: Intel X86 family;
- **RAM**: at least 2GB of RAM;
- **Disk's space**: at least 1GB;
- **Java**: OpenJDK 8 / Oracle JDK 8;
- **Node.js**: Node.js 10.15.1;
- **Maven**: Maven 3.6.0;
- **Browser**: Any browser which support Javascript, HTML5 and CSS3.

### 2.1.3 MacOS

- **Mac Model**: all the models sold from 2011 onwards;
- **RAM**: at least 2GB of RAM;
- **Disk's space**: at least 1GB;
- **Operating system**: OS X 10.10 Yosemite.
- **Java**: OpenJDK 8 / Oracle JDK 8;
- **Node.js**: Node.js 10.15.1;
- **Maven**: Maven 3.6.0;
- **Browser**: Any browser which support Javascript, HTML5 and CSS3.

## 2.2 Configuration

The webserver Tomcat is integrated in the `pom.xml` so you don't need any particular configuration if you are using MacOs or any Linux distro. In Windows you need the set the environment variables check on the setting and add to the "PATH" list the absolute path to the JDK and the Maven bins folders. Usually in Windows, Node.js automatically adds its path to the environment variable.

## 2.3 Execution

To run the backend part, open a terminal or cmd (not PowerShell) in the `Backend` folder, be sure the `pom.xml` is present in the folder, than run the command:

```
mvn clean install
```

The command automatically performs the following actions:

1. Compile the code;
2. Execute test (unit test and static test);
3. Create the executable jar file in the `target` folder.

Once you have completed the build, run the command from the terminal:

```
java -jar target/colletta-*.jar
```

Now Spring Boot is running, to run the frontend just open a terminal window in the "Frontend" folder and run the command:

```
npm start
```

This will automatically open a new browser window with the application frontend.

*We are planning to introduce Webpack dependency to integrate the frontend and backend build life-cycle inside Maven.*

# 3 Workspace Configuration

The purpose of this chapter is to describe the tools used by *SWEight* to develop the application. Obviously, if you are not interested in contributing to this project but you just want to run it, you can use any editor.

## 3.1 IntelliJ IDEA

The default IDE for development is IntelliJ IDEA Community created by Jet Brains, you can use it for Java and JSX development. The commmunity edition used is free and multi-platform, it runs on Windows, MacOs and Linux.

## 3.2 Visual Studio Code

An alternative IDE is Visual Studio Code developed by Microsoft, it's free and open-source and you can use it to write Java and JSX code. There are some plugins created from Pivotal and RedHat which allow you to have an environment similar to IntelliJ IDEA.

## 3.3 Maven

To manage the project you need Maven. It downloads all the dependencies including Spring Boot, compiles the source code and finally runs the application. Maven is written in Java, so you just need the Oracle JDK or the OpenJDK at least version 8.

## 3.4 React

To better debug React components it is recommended to use the following plugins:

- **Mozilla Firefox**(current version 66):
  https://addons.mozilla.org/it/firefox/addon/react-devtools/;
- **Google Chrome**(current version 73):
  https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi

Remember to disable cache in the browser during the development.

## 3.5 Redux

To better debug in Redux it is recommended to use the plugin available at https://extension.remotedev.io/ which can be used as extension in Google Chrome 73 and Mozilla Firefox 66.

# 4 Frontend

This section is intended to make the developer understand the working of the Colletta frontend, and to allow him or her to add functionalities to the software package. In order fully understand the contents below, the developer must have a certain degree of familiarity with React and Redux. If that's not the case, we strongly recommend the reader to at least acquire some basic knowledge on the topics.

## 4.1 Directory tree

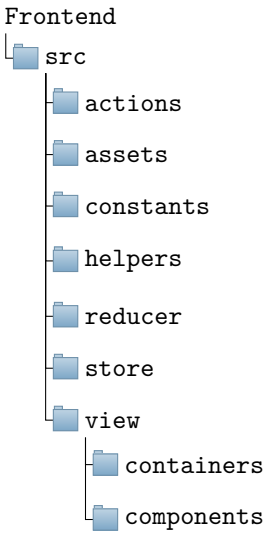


Figure 1: Frontend directory tree

Each folder contains a specific set of files:

- **actions:** the modules in this folder are responsible for creating and dispatching the actions to the reducers;
- **assets:** static files like font and images;
- **constants:** data collections and constants used in various part of the code, i.e. the label used for the translation;
- **helpers:** standard js functions or classes which have some use in the code, i.e. the label translator;
- **reducers:** all the reducers responsible for the creation of a new state;
- **store:** a single file creating and giving access to the centralized state;
- **view:** classes rendering the information in the store. They are divided in *components* and *containers*. The key point to bare in mind when talking about components and containers is the following: containers are "smart", they observe the store and can call actions; components, on the other hand, are basically just static functions.

## 4.2 Modify or add features

### 4.2.1 Components

Components extend the React `component` abstract class and implement the `render()` method. They can be viewed as a pure functions of the props passed by their father component or container. They do not have access to the store. When adding or modifying a component the following rules should be followed:

- Since the global state of the application is managed by Redux, do not use or create the local state of the component. Instead, rely solely on the props;

- Helper functions may be defined in the component class, but none of them should call action creators or external resources such as API calls;
- All components must be placed in the `src/component` folder;
- Every component which needs to render some text must have a language prop to call to the translator module;

When adding a new component, one can start from the following snippet:

```
import React, { Component } from 'react';
import _translator from '../../helpers/Translator';
class SampleComponent extends Component {

  render() {
    const { prop1,prop2,prop3 } = this.props;
    //Do stuff here
    return (
      <React.Fragment>
        {/* Stuff to render */}
      </React.Fragment>
    );
  }
}
export default SampleComponent;
```

### 4.2.2 Containers

Containers extend the React `component` abstract class and implement the `render()` method. They can read the store and alters its state via actions. Containers can also have props passed to them, just like a container. When adding or modifying a component the following rules should be followed:

- Since the global state of the application is managed by Redux, do not use or create the local state of the container. Instead, rely solely on the props and on the store;
- The store and the actions should not be accessed directly for performance and readability reasons. Instead, the should be mapped to the props;
- All containers must be placed in the most appropriate `src/view/containers` sub-directory. Creation of new sub-directories is allowed if necessary.

When adding a new container, one can start from the following snippet:

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import _translator from '../../../helpers/Translator';

class SampleContainer extends Component {

  render() {
    const { prop1, prop2, prop3, action1Prop, action2Prop } = this.props;
    // Do stuff
    return (
      // Stuff to render
    );
  }
}
const mapStateToProps = store => {
  return {
    prop1: store.object1,
    prop2: store.object2,
    prop3: store.object3
  };
};

```

```
const mapDispatchToProps = dispatch => {
  return {
    action1Prop: () => dispatch(action1()),
    action2Prop: () => dispatch(action2())
  };
};
//both action1 and action2 must be imported

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(SampleContainer);
```

### 4.2.3 Rest API calls

When implementing a new Rest API call (RAP from now on), the developer must stick to the following guidelines:

- RAPs must be implemented inside an action creator and should not be put inside components or containers;
- RAPs must use the Axios module;
- RAPs must pass an authorization token (exception made for Login and SignUp) which is kept in the store. The snippet below will show clarify how;
- If a RAP does not need additional data, the data field should be replaced by `{}`, otherwise the response will display a 403 error.

When adding a new RAP, one can start from the following snippet:

```
export const sampleActionCreator = objectToSend => {
  return dispatch => {
    axios
      .post(
        'http://localhost:8081/sample-call',
        {
          ...objectToSend
        },
        {
          headers: {
            'content-type': 'application/json',
            Authorization: store.getState().auth.token
          }
        }
      )
      .then(response => {
        // Maybe do something
        dispatch({ type: 'SAMPLE_ACTION', dataToDispatch });
      })
      .catch(() => {/* Handle Error Here*/});
  };
};
```

### 4.2.4 Interface Language

The multiple languages of the application are implemented via the Translator module in the `src/helpers` folder. Every single piece of text shown to the user is stored in `src/constants/Label.js` in all the languages supported by Colletta. Each language is identified by its ISO 639-1 code.
Each string variable is composed as follow:

context_identifier

where:

- **Context** is the component or container in which the string is used. If it is used in more than one component or container, context should be `gen`. Context must start with a lowercase letter;
- **Identifier** should sum up the function of the string. It must start with a lowercase letter.

The translated string can than be displayed with the following function:

```
_translator('string_variable', language)
```

where:

- **String_variable** is the name of the string you want to render. It must be surrounded by single quotes;
- **Language** is a string containing the ISO 639-1 code of the language you want to render the string in.

Therefore, whenever one wants to add some new text to some component or container, he or she must add the string in all of the supported languages and render it with the translator module.
If instead one wants to implement a new language, the following steps need to followed:

1. Every single string in `Label.js` must be translated and inserted with the ISO 639-1 code of the language;
2. The variable `UiLang` in `src/constants/Label.js` must be updated with the ISO 639-1 code of the language;
3. For every language in `Label.js`, a string int the format `gen_xx` must be added, with xx being the ISO 639-1 code of the language. The string must contain the language name. For instance, in English `gen_it` is Italian and `gen_en` is English.

The following snippet represents a slim-down example of `Label.js`:

```
export const _label = {
  it: {
    account_yourData: 'I tuoi dati',
    dashboard_hiUser: 'Ciao, ',
    executionExercise_complete: 'Completa',
    gen_it: 'Italiano',
    gen_en: 'Inglese'
  },
  en: {
    account_yourData: 'Your data',
    dashboard_hiUser: 'Hello, ',
    executionExercise_complete: 'Submit solution',
    gen_it: 'Italian',
    gen_en: 'English',
  }
};
```

### 4.2.5 Analysis Languages

The process for enabling other analysis language is a little bit more tedious, as it means having to do with the Freeling tagset. Let's have a brief overview at what needs to be done:

1. A custom JSON tagset must be created starting from the freeling tagset;
2. The tagset created must be modified to accommodate some conditions;
3. The tagset must be placed in the correct folder.

One can find the Freeling tagsets and their description at https://talp-upc.gitbook.io/freeling-4-1-user-manual/tagsets. Starting from the tables in the Freelig doc, the end result should look like this:

```
const italian = {
  adjective: {
    text: { full: 'adjective', short: 'A' },
    attributes: [
      {
```

```
        attrName: 'type',
        choices: [
          { short: 'O', full: 'Ordinal' },
          { short: 'Q', full: 'qualificative' },
          { short: 'P', full: 'Possesive' }
        ],
        condition: null
      },
      {
        attrName: 'degree',
        choices: [
          { short: 'S', full: 'superlative' },
          { short: '0', full: 'none' }
        ],
        condition: { index: 1, short: 'Q' }
      },
      //...........
```

Let's break in down:

- The outermost level contains the categories of word in the language (i.e. adjective, noun, etc...);
- Each category has a text value, a name, a condition, and a list of choices. Each choice represents a possible value for the attribute.
- Choices and text objects of attributes contain two string:
  - **Full:** the name of the choice or category (i.e. Adjective, Ordinal).
  - **Short:** the letter corresponding to the choice or the category (i.e. Adjective is A, Ordinal is O.

The modifications which have been made from the standard tagset are the following:

- **Conditions**: this field is needed for avoiding showing the user choices that cannot be taken. For instance, it would make non sense to show the option to show the superlative option if the adjective selected has been marked as possessive. Thus, the condition states that the attribute at `index: 1` must have `short: Q`, so it must be qualificative. If no restriction applies, the condition field must be set to null;
- **None fields**: looking at the code example above, one may notice the following choice: `{ short: '0', full: 'none' }`. It has been added to let the user to mark an adjective as not superlative. This choice must be added whenever an attribute is not strictly mandatory.

Adding conditions and none fields requires a decent understanding of a language grammar, and it is a process that must be executed very carefully.
Once the file is completed, it can be placed in the `src/constants` folder, named as `ger_xx` where `xx` is the ISO 639-1 code of the language.
Disclaimer: Freeling documentation is not 100% reliable. The tagset might be incomplete or containing some errors.

# 5 Backend

This section is intended to make the developer understand the working of the Colletta backend, and to allow him or her to add functionalities to the software package. In order to fully understand the contents below, the developer must have a certain degree of familiarity with Java, the framework Spring Boot with his subframework Spring Data MongoDB, MongoDB and Maven. If that's not the case, we strongly recommend the reader to at least acquire some basic knowledge on the topics.

## 5.1 Directory tree

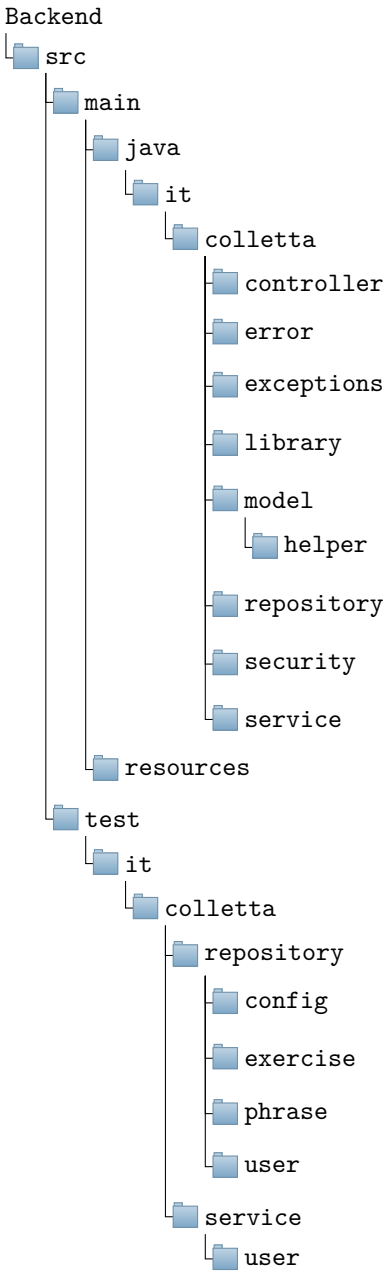


Figure 2: Backend directory tree

Each folder contains a specific set of files:

- **controller:** classes that handle HTTP Request, marked with @RestController Spring annotation;

- **error:** ;
- **exceptions:** ;
- **library:** classes that connect the application to the PoS-tagging library;
- **model:** has the folder *helper* which contains the transfer object (DTO), the other Model files are standard POJO objects that represent the JSON object store inside the database;
- **repository:**;
- **security:** ;
- **service:**;
- **test:** the tests that the application must pass to enter in a safe running state;

# A Glossary