Developer Handbook

SWEight Group - Project Colletta

SWEightGroup@gmail.com

Information about the document

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

Description

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

Change log

Version	Date	Description	Author	Role
1.1.0	2019-05-06	Refactor §5.1	Sebastiano Caccaro	*Writer*
1.0.1	2019-05-06	Fixed §3	Sebastiano Caccaro	*Writer*
1.0.0	2019-04-10	Document approval and release	Damien Ciagola	*Project Manager*
0.4.0	2019-04-04	Checking	Gheorghe Isachi	*Verifier*
0.4.0	2019-04-02	§1 completed	Francesco Corti	*Writer*
0.3.0	2019-04-02	§2 and §3 completed	Francesco Magarotto	*Writer*
0.2.0	2019-04-01	§4 completed	Sebastiano Caccaro	*Writer*
0.1.1	2019-04-01	Written §4 to §4.2.1	Sebastiano Caccaro	*Writer*
0.1.0	2019-04-01	Backbone translated in English	Sebastiano Caccaro	*Writer*
0.0.1	2019-03-18	Document backbone	Enrico Muraro	*Writer*

Contents

List of Figures

1 Introduction

1.1 Document goal

The purpose of this document is to provide all the necessary information to extend, correct and improve Colletta. There will be additional information regarding setting up the development environment to work in an environment that is as consistent as possible with that used by the other members of group SWEight, but can be ignored if you only want to use part of the product. This guide was written taking into account the Microsoft Windows and Linux operating systems. If other systems are used, compatibility issues may arise, even if it's unlikely. In this case refer to the git page. This document will grow as the product will be fully developed.

1.2 Product goal

The purpose of the product is the creation of a collaborative data collection platform where users can prepare and/or perform small grammar exercises. The front-end of the system consists of a web application developed with React and Redux, while the back-end is a Spring Boot application written in Java, which will handle HTTP Requests sent from the front-end.

1.3 References

1.3.1 Installation references

- **Git**: https://git-scm.com/
- **Node.js**: https://nodejs.org/en/
- **NPM**: https://www.npmjs.com/
- **Oracle JDK**: https://www.oracle.com/technetwork/java/javase/downloads/index.html
- **OpenJDK**: https://openjdk.java.net/
- **Maven**: https://maven.apache.org/
- **Lombok**: https://projectlombok.org/
- **VSCode**: https://code.visualstudio.com/

1.3.2 Legal references

- **MIT License**: https://opensource.org/licenses/MIT

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 supports 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 supports 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 supports 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 distribution. 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, open a terminal or cmd (not PowerShell) in the `Backend` folder, be sure the `pom.xml` is present in the folder, then 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 install npm start
```

1. `npm install` will download all the dependencies needed to execute the code;
2. `npm start` will run the development server;

At the end a new browser window will be opened and the frontend will be loaded.

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. 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 community edition 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 by 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. You can download it at the link Maven page

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/

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 Architecture

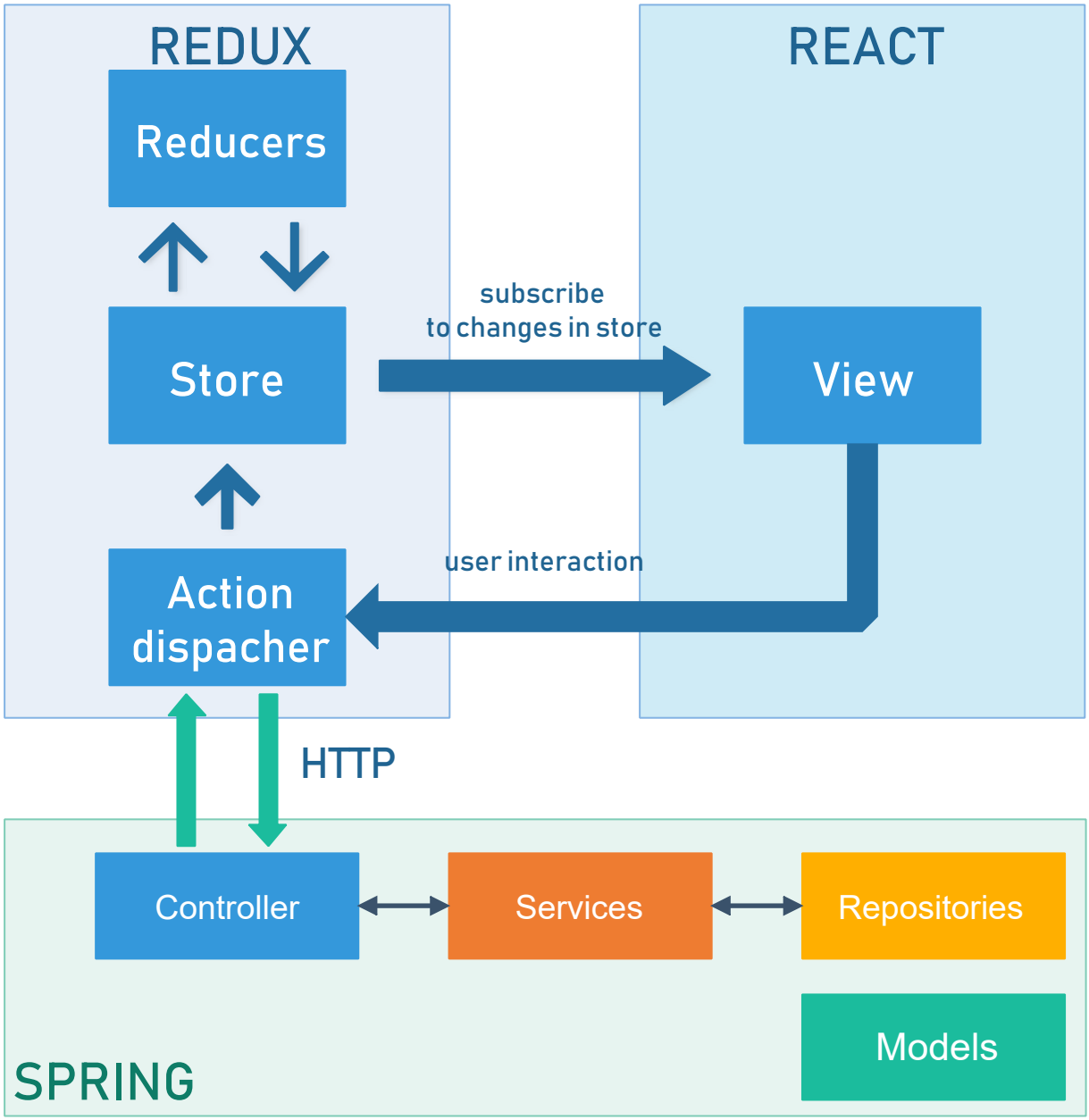

Figure 1: General schema of the application

Figure 1 contains the overview of the application architecture. The software package uses React-Redux for the Frontend which communicates with a Springboot based Backend via REST calls.
A more detailed view at the architecture can be found in subsection 5.1 and @@@@@@@@@@@@Riferimento architettura backend@@@@@@@@@@@@.

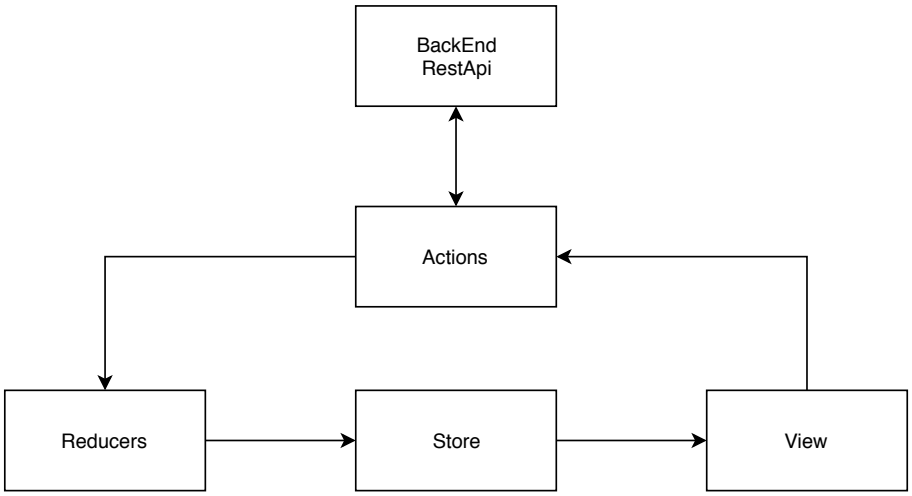

Figure 2: Basic Redux architecture

5 Frontend

This section is intended to make the developer understand the inner workings of the Colletta frontend, and to allow him or her to add more functionality 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.

5.1 Architecture

In Figure 3, one can have a glimpse into the general architecture of the Frontend of the software package. Although it may seem quite complex at first, the architecture is actually just a more expanded version of the basic Redux patter in Figure 2:

- Reducers, Actions and Views are grouped by logical function to increase code readability and reduce pairing;
- Helpers and Constants are declared to deal with minor logic which cannot be dealt with in the Backend.

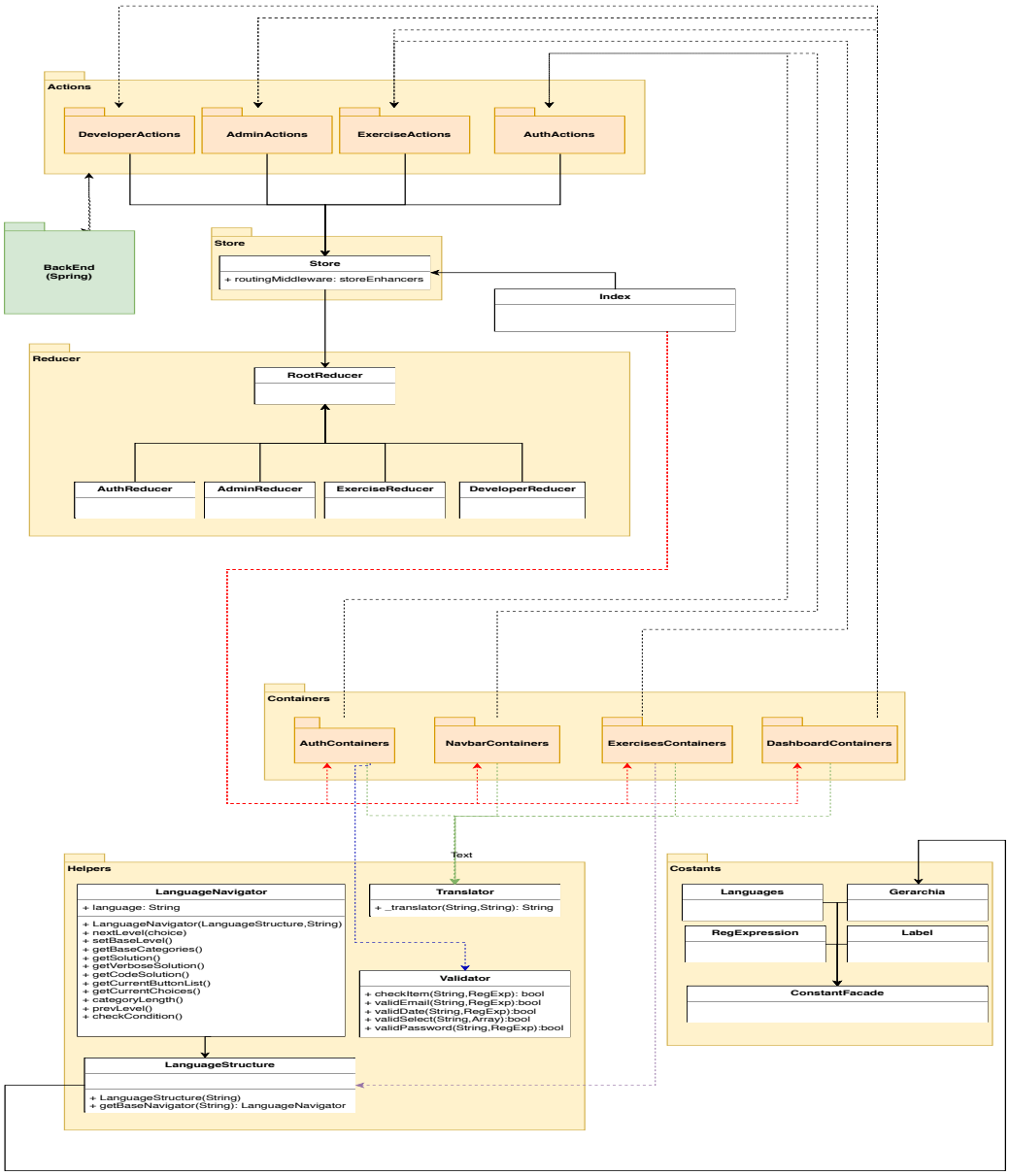

Figure 3: React and Redux architecture

5.2 Directory tree

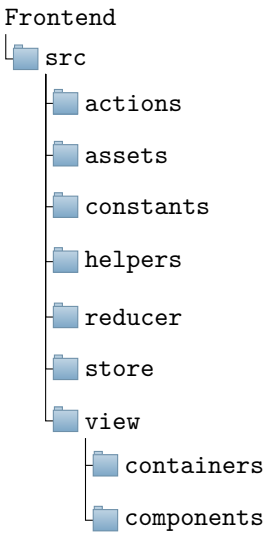

Figure 4: Frontend directory tree

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 parts 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 bear 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.

5.3 Modify or add features

5.3.1 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;
```

5.3.2 Containers

Containers extend the React `component` abstract class and implement the `render()` method. They can read the store and alter 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, they 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);
```

5.3.3 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 Log-in and Sign-up) 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*/});
  };
};
```

5.3.4 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 be 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 in 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',
  }
};
```

5.3.5 Analysis Languages

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

1. A custom JSON_G 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_G tagsets and their description at: freeling-4-1-user-manual. Starting from the tables in the Freeling 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: 'Qualifying ' },
        { short: 'P', full: 'Possessive' }
      ],
      condition: null
    },
    {
      attrName: 'degree',
      choices: [
        { short: 'S', full: 'superlative' },
        { short: 'O', 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 to avoid showing the user choices that cannot be taken. For instance, it would make no sense to show the option to select 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 qualifying. 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: 'O', 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 the grammar of a language, 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 always reliable. The tagset might be incomplete or contain some errors.

6 Backend

This section is intended to make the developer understand the inner workings of the Colletta backend, and to allow him or her to add more functionality 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 sub-framework 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.

6.1 Directory tree

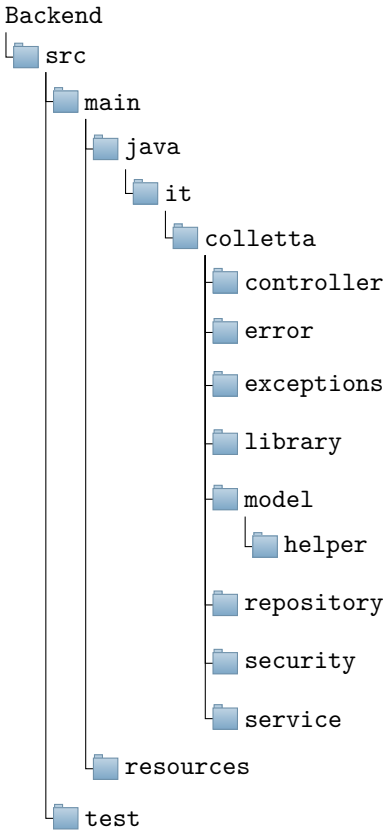

Figure 5: Backend directory tree

Each folder contains a specific set of files:

- **controller**: classes that handle HTTP Request, marked with *@RestController* Spring annotation;
- **error**: custom classes for error handling;
- **exceptions:** custom classes for exception handling;
- **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 JSON objects store inside the database;
- **repository**: classes that encapsulate the set of objects persisted in a data store, in our case MongoDB, and the operations performed over them, providing a more object-oriented view of the persistence layer;
- **security**: classes that manage the security of the application through encryption and token management;
- **service**: classes that make up the logical business part of the application;

- **test**: the tests that the application must pass to enter in a safe running state;
- **resources**: contains the configuration file for the connection to the database.

6.2 Data and logic separation

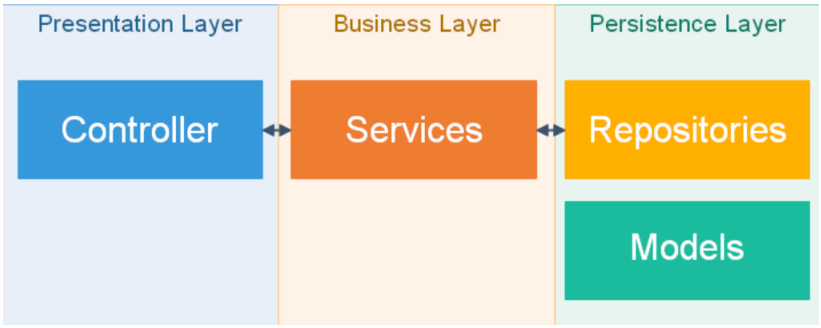

Figure 6: Data and logic separation

The design pattern used for the backend is: **Spring MVC**. Respectively with the Controller, Service and Repository classes.
More information about the architecture chosen for the backend is available at the link: Spring Web MVC.

6.3 Security

The system was designed to secure client server communications so that only a user registered in the system can make calls which are then resolved by interacting with the database.
In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned. Instead when a user registers for the first time the token will be generated.
Whenever the user wants to access a protected route or resource he/she must necessarily send his/her token that identifies he/she has correctly signed-in. If this is not done the system will decline the request.
The security system has been implemented so that the only requests that are authorized without token recognition are the login (*/login*) and the registration (*/sign-up*).
The system uses *JWT tokens* for token authentication, more information is available at the link: Introduction to JSON Web Tokens.

6.4 Modify or add features

6.4.1 Controller

The controller has the annotation *@RestController* above the class definition and contains the methods that map the HTTP calls sent from the frontend, being handled by the `DispatcherServlet`. *@RestController* is a convenience annotation which contains within it *@Controller* and *@ResponseBody*.
This configuration returns to the frontend a JSON file, if this does not want to be done the @RestController word must be replaced by *@Controller*.
When adding a controller the following rules should be followed:

1. The new controller must have the annotation *@RestController* above its definition;
2. Each mapping must be unique in the application. If not, Spring throws a `RuntimeException` during application initialization. You cannot use parameters to differentiate your endpoints;
3. The name of the controller should be significant;
4. The controller must **only handle HTTP Requests**, any kind of input control must be performed in the service that will then be called;
5. The services should be instantiated with the *@Autowired* keyword, see Guide to Spring @Autowired for more information over all the methods.

6. Each method must contain the signature: @RequestMapping (value = "/", method = Request.RequestType)

When adding a new *Controller*, one can start from the following example:

```
@RestController
public class UserController{

  @Autowired
  UserService service;

  @RequestMapping(value = "/exercises/student/do", method = RequestMethod.POST,
    produces = MediaType.APPLICATION_JSON_VALUE)
  public ResponseEntity<Object> doExercise(){
      //call a generic service
  }
}
```

6.4.2 Service

Service has the annotation *@Service* to indicate that the class holds the business logic.
The service takes care of **checking** and **validating** the **inputs** sent by the frontend, if incorrect data is sent the service takes care of generating an exception that will be managed by the *Controller* class.
When adding a service the following rules should be followed:

1. A service can only call its repository;
2. All database read and write operations must be performed from the repository

When adding a new *Service*, one can start from the following example:

```
@Service
public class UserService{

  @Autowired
  UserRepository userRepository;

  public List<String> findAllExerciseToDo(String userId){}
}
```

6.4.3 Repository

The repository classes manage the persistence layer. These classes are marked with `@Repository` annotation and they implement the interface `MongoRepository`. Implementing the MongoRepository interface allows you to call the CRUD query interface. If you want to write custom queries, you have to create an interface and put the method's signature inside, then you have to create a concrete class which has the "Impl" suffix to its name. The repository class should extend the MongoRepository and the custom query interface, if present.

6.4.4 Model

The Model classes are marked with the *@Document* annotation to indicate that these classes are mapped in the database. The annotation can specify the name of the collection of the database in which the model will be stored in this way:
@Document(collection = "mycollection")
A field annotated with *@Id* will be mapped to the '_id' field in the database.
When adding a Model the following rules should be followed:

1. The model's name must be significant.

When adding a new *Model*, one can start from the following example:

```
1
2  @Document(collection = "phrases")
3  public class PhraseModel {
4    @Id private String id;
5
6    @Indexed(unique = true)
7    private String phraseText;
8
9    @Builder.Default private ArrayList<SolutionModel> solutions = new ArrayList<>();
10   private String language;
11   private Long datePhrase;
12
```

A Glossary

F

Freeling: the library for pos-tagging developed by TALP Research Center written in C++;

P

Pos-tagging: part-of-speech tagging, also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech;
POJO: Plain Old Java Object, is an ordinary Java object, not bound by any special restriction and not requiring any class path. In Spring it refers to a Java object (instance of definition) that isn't bogged down by framework extensions;

J

JSON: JavaScript Object Notation, is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.
JWT: JSON Web Token, a JSON-based open standard (RFC 7519) for creating access tokens that assert some number of claims;