UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

COS301 Capstone Project

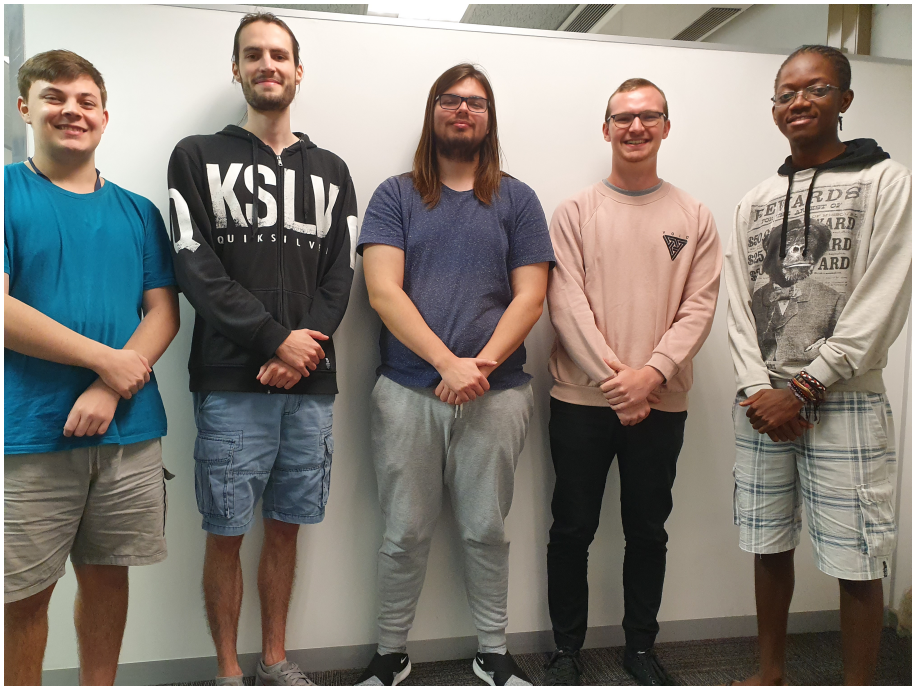
Team Syntactic Sugar

Jargon Sentiment Analysis Testing Policy

Team Members
Graeme Coetzee
Christiaan Nel
Ethan Lindeman
Kevin Coetzee
Herbert Magaya

Client
Compiax

May 23, 2019



Contat email: syntacticsugar9@gmail.com

Contents

1 Introduction

1.1 Purpose

This *Coding Standards Document* is intended to be a guide on the policies, standards and practices that are followed when working on the *Jargon* project. The remainder of this section describes the different services used within the project and the remainder of this document then continues to describe the coding standards applied to each individual service.

1.2 Repository Structure

```
/
├── services/
│   ├── frontend/
│   │   └── [...]
│   ├── neural-network/
│   │   └── [...]
│   ├── listeners/
│   │   └── [...]
│   └── controller/
│       └── [...]
├── documentation/
│   ├── publish/
│   │   └── [...]
│   ├── <document-name>/
│   │   ├── <document-name>.tex
│   │   └── images/
│   │       └── [...]
│   └── [...]
└── [...]
```

The Jargon Sentiment Analysis project makes use of a `monorepo` structure for our Git repository. The repository is also structured to reflect our Microservices architectural style. The repository contains 2 main folders on interest: `services` and `documentation` (see 6).

There are four (4) folders within the `services` folder, each representing a service in our Microservice architecture: `frontend`, `neural-network`, `listeners` and `controller`. Each service will contain their respective *README.md* files. A short description of the contents of each folder is listed below.

- `frontend`

  Contains a web application used for creating and managing Sentiment Analysis projects. The web application is built using *Angular 7*.

- `neural-network`

  Contains the neural network service used by the for analysing sentences and returning their sentiment values. The Neural Network is built using *Python* and the *PyTorch* library.

- `listeners`

  Contains the listeners service that retrieve sentences from an online source, such as Twitter. The listeners is built using *Node.js* and the *Express.js* and *mongoose* libraries.

- `controller`

  Contains the controller service that will create and edit projects. The controller is built using *Node.js* and the *Express.js* and *mongoose* library.

2 Global Standards

The following standards are applied to all files across all folders, with the exclusion of automatically generated files.

2.1 File Format

- Files are encoded using the **UTF-8** character set.
- Lines should not be longer than **80 columns**.
- **Soft tabs** expanded to **4 spaces** should be used.
- Each level of indentation uses **1 tab**.
- Line continuation indentation uses **2 tabs**.
- Every file contains a **file header**.

2.2 File Header Layout

Headers are always on the first line of a file and are placed in comments. The following information must always be present where applicable:

- Name of the file
- Original author of the file
- Name of the class(es) contained within the file
- Short description of the file

The file header layout is illustrated below. The `(start)` and `(end)` tags indicate the block-comment start and end symbols. As these are language specific, they are described in the sections dealing with per-language standards.

```
(start)
    Filename: File.ext
    Author  : John Doe
    Class   : SampleClass

        The SampleClass contains many different sample
        methods.
(end)
```

3 Front End Standards

This section describes the coding standards applied to the `frontend` folder. The system design of the subsystem is illustrated in Section 3.1 and the file structure of the folder is explained in Section 3.2. The web application makes use of TypeScript, HTML and CSS. The standards for these languages are described in Sections 3.3 and 3.4.

3.1 System Design

System Design (if needed)

Figure 1: UML Class Diagram of the Web Application

3.2 File Structure

```
services/
└── frontend/
    ├── src/
    │   ├── app/
    │   │   ├── components/
    │   │   │   └── [...]
    │   │   ├── services/
    │   │   │   └── [...]
    │   │   ├── interfaces/
    │   │   │   └── [...]
    │   │   └── [...]
    │   ├── assets/
    │   │   └── [...]
    │   └── [...]
    └── [...]
```

The `frontend` folder contains Angular CLI generated files and folders used for deploying and compiling of the service.

The most important folder is the `src/` folder, which contains the following notable locations:

- `app/`

  This folder contains all of the source code of the web application. The most important files here are `app.component.ts` and `app.module.ts` which describe the root Angular components.

- `app/components/`

  This folder contains different subfolders. Each subfolder represents one of the components as represented in Section 3.1. Each subfolder contains the `.html` and `.ts` files associated with each component.

- `app/interfaces/`

  This folder contains the `.ts` files that describe the enums and interfaces used in the web app.

- `app/services/`

  This folder contains different subfolders. Each subfolder represents one of the services as represented in Section 3.1. Each subfolder contains the `.ts` files associated with each service.

- `assets/`

  This folder contains all the assets such as background images and icons used within the web app.

3.3 Typescript Standards

3.3.1 Naming Conventions

- **Variables** are named using camel casing. Descriptive names should be used with the exception of counters in loops.
- **Classes** start with a *capital* letter and use camel casing.
- **Components** must have the word `Component` as the last word in the class name. Similarly, **Services** must have `Service` as the last word.
- **Functions** are also named similar to regular variables and should be descriptive.

```
export class SampleComponent {

    sampleVariable : string;

    public myFunction() : void {
        ...
    }
}
```

3.3.2 Style

Braces will be styled in the following manner:

- Opening braces are placed on the same line as the header.
- Closing braces are placed on a separate line at the same indentation level as the header.
- Else clauses are placed on the same line as the closing brace.
- While clauses of a do-while are placed on the same line as the closing brace.
- Braces are never left out for one-line loops or conditions.

```
if (condition) {
    statement;
} else {
    statement;
}
```

```
while (condition) {
    statement;
}

do {
    statement;
} while (condition);
```

Continuation lines should end on the operator as to indicate that the line is not complete and has a continuation.

```
var result = example + of + (a * very) /
        long - equation;
```

3.3.3 Comments

File headers are structured according to section 2.2 and are styled in the following way:

```
/***
 * Filename: sample.component.ts
 * Author  : John Doe
 * Class   : SampleComponent
 *
 *     The SampleComponent contains many different sample
 *     methods.
 ***/
```

Function headers are provided for every function and take the following form (the description can be omitted in the case of simple functions, such as mutators & accessors).

```
/***
 * function(ParType1, ParType2) : ReturnType
 *
 *     Description of the function.
 ***/
function(p1 : ParType1, p2 : ParType2) : ReturnType {
    ...
}
```

Inline comments should be kept to a minimum, since code should be self-documenting. Only use inline comments in the case of code that may be difficult to understand.

3.4 HTML & CSS Standards

3.4.1 Style

CSS files should be styled in the following way:

```
selectors {
    some-attribute: style;
    another-attribute: style;
}
```

```
more {
    some-attribute: style;
}
```

HTML files should be styled according to the following rules:

- Opening and closing tags of **block** elements should be kept on their own lines, with the content indented.
- Opening and closing tags of **inline** elements should be kept on the same line, with the content between the tags.

```
<div>
    <p>
        Here is some <span class="red">red</span> text!
    </p>
</div>
```

3.4.2 Comments

File headers are structured according to section 2.2 and are styled for CSS and HTML, respectively, in the following ways:

```
/***
 * Filename: style.css
 * Author  : John Doe
 *
 *     The styling for some example page is contained
 *     here and applies a material style.
 ***/
```

```
<!--
    Filename: page.html
    Author  : John Doe

        The page displays some content.
-->
```

4 Neural Network Standards

This section describes the coding standards applied to the `neural-network` folder. The system design of the subsystem is illustrated in Section 4.1 and the file structure of the repository is explained in Section 4.2. The Neural Network services makes use of Python and its PyTorch library. The standards for the Python language are described in Section 4.3.

4.1 System Design

System Design (if needed)

Figure 2: UML Class Diagram of the Neural Network

4.2 File Structure

```
services/
└── neural-network/
    ├── server.py
    ├── swagger.yml
    ├── operations.py
    ├── neural-network.py
    └── [...]
```

The root of the `neural-network` folder contain the following files:

- **server.py** that act as the main program for the Neural Network service.
- **swagger.yml** that describes the structure of the API service.
- **operations.py** that defines the functions to be call for each API endpoint.
- **neural-network.py** that defines the Neural Network class and its functions.

4.3 Python Standards

4.3.1 Naming Conventions

- **Variables** is a lower case word, or words separated by an `underscore`. Descriptive names should be used with the exception of counters in loops.
- **Member variables** are named similarly to regular variables with the addition of an *underscore* prefix.
- **Classes** start with a *capital* letter and use camel casing.
- **Functions** are also named similar to regular variables and should be descriptive.

```
def my_function_two(num_one, num_two):
    return num_one + num_two

class SampleClass:
```

```
    _some_member
    _value

    def my_function(self, number):
        self._some_integer = number
        self._value = self._value * self._some_integer
```

4.3.2 Style

The standard Python **indentation** is used to determine the grouping of statements:

```
if (condition):
    statement
else:
    statement

while (condition):
    statement
```

Continuation lines should end on the operator as to indicate that the line is not complete and has a continuation. Line continuation should use Python's implied line continuation inside parentheses, brackets and braces, unless long multiple with-statements are used which cannot use implicit continuation, so backslashes should be used. Both are shown below:

```
result = (example + of + (a * very) /
        long - equation)

with open('/path/to/some/file/you/want/to/read') as file_1, \
        open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

4.3.3 Comments

File headers are structured according to section 2.2 and are styled in the following way:

```
"""
    Filename: some-python-file.py
    Author  : John Doe
    Type    : Class or Functions

        The some-python-file.py contains many different sample methods
"""
```

Function & class headers are provided for every function and class take the following form (the description can be omitted in the case of simple functions, such as mutators & accessors). The function & class headers should appear after its definition with an indentation of 4 spaces.

```
def some_function(param_one, param_two):
    """
        some_function(paramType1, paramType2) : returnType
            Description of the function.
    """
```

```
class SomeClass():
    """
        Description of the class.
    """
```

Inline comments should be kept to a minimum, since code should be self-documenting. Only use inline comments in the case of code that may be difficult to understand.

5 Controller Standards

This section describes the coding standards applied to the `controller` folder. The system design of the subsystem is illustrated in Section **??** and the file structure of the repository is explained in Section 5.1. The controller makes use of Node.js. The standards for the JavaScript language are described in Section 5.2.

5.1 File Structure

```
services/
└── controller/
    ├── db/
    ├── routes/
    ├── models/
    └── server.js
└── [...]
```

The `controller` folder contains the following files and folders:

- **db/** This folder contains a database connection file, and a database configuration file.
- **routes/** This folder contains different route *JavaScript* files. Each file defines and describes an endpoint for the `controller-api` service.
- **models/** This folder contains different schema *JavaScript* files. Each file defines the schema for an object to be saved to the database.
- **server.js** This file defines the main starting point for the controller

5.2 JavaScript Standards

5.2.1 Naming Conventions

- **Variables** are named using camel casing. Descriptive names should be used with the exception of counters in loops.
- **Member variables** are named similarly to regular variables with the addition of an *underscore* prefix.
- **Classes** start with a *capital* letter and use camel casing.
- **Functions** are also named similar to regular variables and should be descriptive.

```
class SampleClass {
    var _someMember;

    function myFunction() {
        let someInteger;
    }
}
```

5.2.2 Style

Braces will be styled in the following manner:

- Opening braces are placed on the same line as the header.
- Closing braces are placed on a separate line at the same indentation level as the header.
- Else clauses are placed on the same line as the closing brace.
- While clauses of a do-while are placed on the same line as the closing brace.
- Braces are never left out for one-line loops or conditions.

```
if (condition) {
    statement;
} else {
    statement;
}

while (condition) {
    statement;
}

do {
    statement;
} while (condition);
```

Continuation lines should end on the operator as to indicate that the line is not complete and has a continuation.

```
let result = example + of + (a * very) /
        long - equation;
```

5.2.3 Comments

File headers are structured according to section 2.2 and are styled in the following way:

```
/***
 * Filename: sample-file.js
 * Author  : John Doe
 *
 *     The sample-file file contains many different sample
 *     methods.
 ***/
```

Function headers are provided for every function and take the following form (the description can be omitted in the case of simple functions, such as mutators & accessors).

```
/***
 * function(ParType1, ParType2) : ReturnType
 *
 *     Description of the function.
 ***/
```

```
function(p1, p2) {
    ...
}
```

Inline comments should be kept to a minimum, since code should be self-documenting. Only use inline comments in the case of code that may be difficult to understand.

6 Documentation Standards

This section describes the standards of the `documentation` folder. As there are no strict standards for the layout of the LaTeXfiles, only the file structure of the folder is described in Section 6.1.

6.1 File Structure

```
/
├── documents/
│   ├── coding-standards/
│   │   └── [...]
│   ├── requirements/
│   │   └── [...]
│   ├── testing-policy/
│   │   └── [...]
│   └── user-manual/
│       └── [...]
└── [...]
```

The repository contains a folder for each of the four documents, namely `coding-standards/`, `requirements/`, `testing-policy/` and `user-manual/`. Each folder contains their respective `.pdf` file of the document.

Version control for the each document are hosted on Overleaf, to allow real-time collaboration.

7 Code Review

This section describes how and when code is reviewed. It further describes who is responsible for reviewing code.

7.1 Review Process

When a team member creates a pull request to merge the development `branch` into `master`, they should assign atleast two (2) other members to review the pull request. The code reviewers will review the code through inspection by looking at the changes proposed in the pull request. If the code complies to the repository's standards as described in this document and the build succeeds, the pull request is accepted. After atleast two (2) members accepted the pull request, the `development` branch is merged into `master`.

If the code does not comply to the repository's standards, the person responsible for the code is notified and must amend the pull request to change the code. Once the code complies to the standards, the pull request is then accepted.