Architectural Requirements Document - Try Catch Degree - TensorFlowUI

Try Catch Degree

August 2021

Name	Surname	Student Number
Felipe	Kosin Jorge	u17291195
Werner	van Rensburg	u15118046
David	Walker	u19055252
Siviwe	Lechelele	u18221409
Wessel	Kruger	u18014934

1 Architectural Design Strategy

When deciding on an architectural design strategy, we were forced to pick between three options:

- Decomposition
- Quality requirement driven design
- Generation of test cases

Ultimately, we decided to use our quality requirements to dictate our design decisions. This is to try to optimise and keep in mind our end goal for the system as a whole, and ensure that the main thrust of our work was always centered around the completion of said quality requirements, without allowing ourselves to be distracted.

Specifically, test cases would be troublesome to design around given that our program is intended to be highly interactive and based on user input, which would mean that the test cases which would account for one user's use cases would be entirely different from another's, and thus it would be very challenging to use said tests to create a cohesive and focused design which doesn't extend its scope too far.

Decomposition is an interesting strategy which could be quite useful in dissecting how the system should fit together, but upon examining the nature of our system and its relatively tightly-put-together systematic style, we realized that it would result in far too vague a total design, and leave us somewhat lost when making decisions. Hence, the decision to focus on quality requirements and make choices which optimise those outcomes was made.

2 Architectural Styles

2.1 Decisions

This project uses a combination of multiple different architectural styles to achieve the total project result. Firstly, within the web application itself, it uses an **object-oriented**, **component-based and layered** architectural style, as is in line with the Angular framework which said web app is built around.
In addition, when extended to run code externally, the program uses the **client-server** style, in which the web app calls upon a Python-based Flask server to run generated code and return an output. The same style is used to access and store objects in a cloud-esque community library.

2.2 Rationale

These decisions were made for a variety of reasons. Firstly, the layered, object-oriented nature of the program was chosen simply as a good principle and good fit with the Angular framework with which the client requested we operate. Hence, by choosing to follow established routes with said framework, we could optimise the time used to extend upon it, and create a cohesive final product.
The Client-Server Architectural pattern is a part of the Distributed Systems Style. Since our system is distributed between the client and the server when running code or saving to a cloud-like library, this style fits perfectly. This style was also chosen to facilitate the communication between the client and the server. Since the client layer won't need the server layer to run properly, the client side will only make requests do the server when the user specifically requests it. This adds a layer of security to the system as well as diminish the amount of data transmitted.

3 Architectural Quality Requirements

3.1 Testability

3.1.1 Project Testing

The user must be able to test the entire system as a whole and any code generated by the system at any given time. In this vein, the project needs to have built-in testing functions wherever possible, which can be run by automated testing software to ensure maximum functionality with minimal testing effort.

3.1.2 Node Testing

The user must be able to individually test created functionality and individual nodes at any time, including any node's interaction with given child nodes. The user must be able to test generated code for a single node.

3.1.3 Multiple Node Testing

The user must be able to test the functionality of a group of nodes or merged nodes at any given time.

3.2 Performance

3.2.1 Load times

The program should open in no more than 125 percent of the time it takes to open only a web browser. For example, on a computer which takes 2 seconds to open a web browser, the TensorFlow UI should take no more than 2.5 seconds to open.
Furthermore, the code generation algorithm shouldn't require additional sub-routines above and beyond a simple generation step for each node, so in the worst case code generation from our node tree structure should occur in O(n) time.

3.2.2 Interaction performance

Interactions with the program's menus and controls should be practically instantaneous, meaning that all menus should appear within half a second of being clicked, and all new screens should do the same. Furthermore, animations should be fluid, smooth enough to be perceived as movement to the human eye.

3.2.3 Resource usage

The program should not consume more than 500MB of RAM when open.

3.3 Usability

3.3.1 Learnability

They system should be intuitive to the user, the user should at no time not know what to do next. To this end, each interface component should in some way be labelled descriptively.

3.3.2 Efficiency

The system needs to be structured in a way that the logical steps after a process has been completed is obvious to the user. That is, there should be visual feedback for all actions taken.

3.3.3 Memorability

The system needs to be memorable to the user. The user does not need to relearn how the system works. The system needs to remember the user and their current working state.

3.3.4 Satisfaction

The system needs to be designed in a way that exploring the rest of the system is enjoyable to the user.

3.4 Scalability

We do not expect multiple users to work on the same system since users will be downloading the UI and working from their own computer. Therefore scalability for this project will be examined in terms of project size and concurrent saves to the community library:

3.4.1 Project Size

Each individual project can have a different amount of nodes. This means that one project may have 10 nodes and another may have up to 100 nodes.
As such it is important that the chosen architecture behind the project building aspect of the TensorFlow UI can allow for scaling to keep up with the different sizes of projects. Any user should be able to perform equal levels of computation to an equivalent TensorFlow Python application. this end, no hard caps should be imposed on a maximum number of users.

3.4.2 Community Library

This will be a single library that will store projects from multiple users. We expect 50 initial users, but as the project popularity grows, it will increase from 10 to 100 and eventually 1000+.
As such it is important that the chosen architecture can allow for vertically scale-out to ensure the the demand for saving and accessing the community library can be met. To this end, no hard caps should be imposed on a maximum number of users.

3.5 Maintainability

Maintainability is centered around how easy or difficult it is for the system to comply with standards or conventions regarding maintainability, There are four main sections which are looked at to determine the Maintainability, namely: analysability, changeability, stability and testability. Some of these have already been discussed.

3.5.1 Maintainability of the System

It is difficult to quantify or be able to easily verify that a system is actually maintainable, one measure that can used to test for it is 'The complete fitting function', the higher this number is, the more maintainable a system is deemed to be. This is measured using the formula:

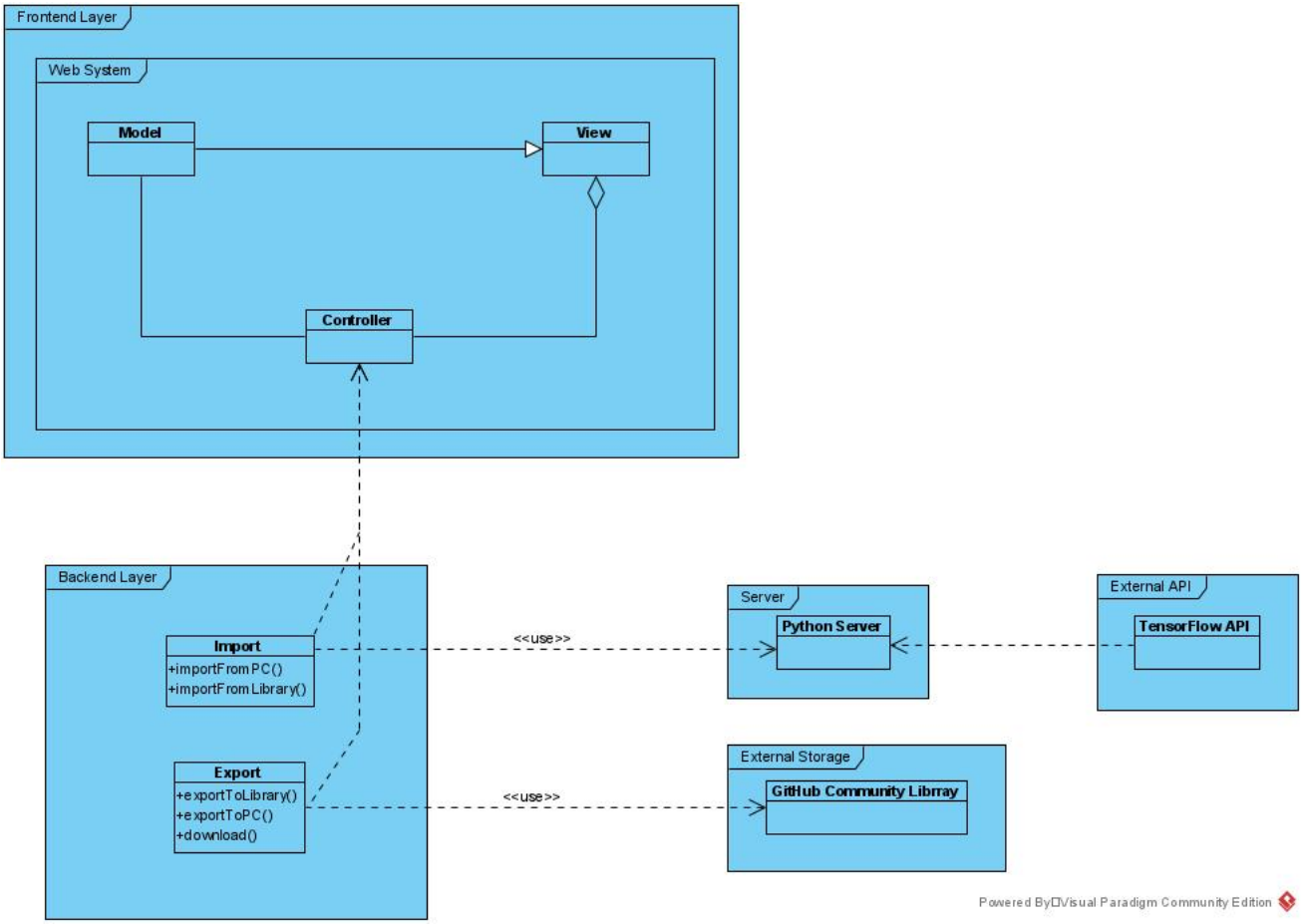
3.5.2 Code Duplication

Excessive amounts of duplication are detrimental to the systems maintainability, the duplication of code affects the analysability and changeability of the system. For our system, we will aim to have the overall code duplication in the code to be less than 8%, as this would be considered as well designed according to the maintainability standards. and changeability.

3.5.3 Unit Testing

Having good and extensive unit testing has a significant impact on the maintainability of the system. Unit tests raise testability, stability (as they ensure the system does not break at any point) and they also improve the analysability of the system. Thus for each use case, we expect the system to have unit test coverage of 80% or more of the use case.

4 Architectural Design and Pattern



The Design pattern shows a two layer architecture due to our chosen Client-Server architecture style, which are the Client Layer and the Backend Layer. The Backend layer consists of import and export functionality, as well as a TensorFlow controller which maps TensorFlow functionality to each Node in our system. The Front layer consists of our Web application which has a MVC (Model-View-Controller) style for increase security, functionality and a simpler component based system.

5 Architectural Constraints

The aim of the TensorFlow UI application is simplicity and speed. Hence, the architectural constraints tend to revolve around those particular factors, which both inform the decision to implement the UI as essentially no more than a progressive web app (PWA). Furthermore, the application is entirely open source, as it aims to be recognised at some point as the "official" UI of the TensorFlow machine learning framework. Thus, the following contraints are particularly relevant:

- Low download and installation size, to the point that it can easily be used by practically any computer on any internet connection.

- Deployability to web servers and accessibility through web browsers once deployed.
- The use of only open-source frameworks and codebases to ensure free use and access without licensing restrictions.

Furthermore, the client specified certain technologies which have to be used, within which we are constrained:

- Angular with various additional libraries and framework integrations for the frontend app creation.
- Python for any potential backend layers which will be used to run generated TensorFlow code.

For more details on the technical choices made, please refer to the technology choices below.

6 Technology Choices

For the frontend, our system requires a well designed web interface. This could be achieved by using JavaScript and HTML. Raw HTML and JavaScript is however insufficient for what the frontend should illustrate. One constraint and the most important aspect of this system is to easily visualize code using a graph or components that corresponds to certain code blocks. The interface should be simple yet efficient.
Because of this constraint, we decided to use Angular. Angular allows us to build the needed visual blocks to illustrate code. It allows us to connect these blocks and add information (like types and variable values) in the frontend itself. Since nodes in the frontend already contain all the information and connections it needs to operate, all the backend needs to do is to use this information to generate code.
Furthermore, Angular has a number of useful importable libraries and modules which enable extensibility of the platform beyond what typical, framework-free JavaScript and HTML would provide. Specifically, we will use Angular Material, which follows Google's Material guidelines for a responsive and visually pleasing user interface.

The backend needs to be simple, reusable, maintainable and testable. Our system is by no means the end product of what a TensorFlow UI should be capable of and therefore the backend (as well as the frontend) should be modular so that new functionality can easily be added to the existing system.
The backend exists in two forms: firstly, as a community library, which is hosted on GitHub's servers, and accessed through simple API calls, and secondly as a code sandbox, which users can test generated code in. This sandbox is run in a Docker container, complete with a running Python server program which uses the Flask library to provide an endpoint for code to be submitted to, after which it is run and the output then returned.

6.1 Pros

- Angular with Angular Material components allows easy creation of various visual effects and graphs needed in the interface of our system.
- Python is quick and easy to implement.
- Documentation and support for Angular is widely available, which means that any developer wishing to add more functionality to our system will not struggle to understand the framework used.
- Most of the system's functionality is handled within theAngular components of the frontend. This means that the system is extremely lightweight and does not have extreme hardware requirements.
- The GitHub storage is tightly integrated with the platform we use for the rest of our project management.
- Angular Material offers modern, visually pleasing components with relatively little additional effort over using unstyled or poorly-styled generic components.

6.2 Cons

- Communication between the frontend and backend is somewhat limited and a more complex strategy is required to allow the frontend and backend to communicate.
- Building and adding new components to the frontend and their required functionality can only be done in Angular.
- Angular moves quickly and often requires relearning of previously-known paradigms.