Architectural Design Document

Ctrl Alt Defeat

June 26, 2023

1 Architectural Design Strategy

1.1 Approach

We have opted to follow the approach of: Design based on Quality Requirements

1.2 Reasons for approach

- This approach allows us to model our system as a solution to an abstract problem (that being the satisfaction of quality requirements). This allows us to make technology-independent design decisions.
- By formulating our design based on quality requirements, we are able to ensure (make a huge step towards ensuring) that the system does indeed meet its quality requirements first and foremost before any code has been written.
- By analyzing the system in terms of quality requirements, it becomes clearer which architectural strategies and patterns (discussed below) are most suitable for the system's implementation.

1.3 Reasons to not follow other approaches

- Since our application is dashboard centric, which limits edge case interaction from users, we found it unsuitable to take the approach of designing the system via the generation of test cases. Furthermore, designing test cases would not necessarily help us in identifying approriate architectural stategies and patterns (at least not the the extent that our favoured approach allows)
- There are relatively few object-like structures that can be decomposed into suitable hierarchical or relationship structures (at least not to enough of an extent) in our system. Hence we felt that a decomposition approach was not necessarily warranted (at least not over our favoured approach, which we deemed to be the most suitable for our purposes)

2 Architectural Quality Requirements

2.1 heading

Testing out LaTeX

3 Architectural Strategies

We have adopted aspects of several architectural styles, patterns and strategies to help realise our system and its quality requirements.

3.1 heading

Testing out LaTeX

4 Architectural Design and Patterns

4.1 heading

Testing out LaTeX

5 Architectural Constraints

5.1 Client-Defined

- The system needs to consist of at least two seperate deployable units.
- The system must be deployed to a private virtual machine with a public IP address (provided by Southern Cross Solutions). This machine is subject to extremely frequent cyber-attacks for which the system must protect against.
- The system must make use of at least one NoSQL document-based database

5.2 Hardware and Operating System Constraints

- The system is to be hosted on a single virtual machine (running on a single physical server). At a later date this may be expanded to make use of multiple virtual machines running on different physical machines.
- The system must be suitable for and run on an Ubuntu-style Linux operating system

6 Technology Choices

6.1 Frontend Technology

Testing out LaTeX

6.2 Backend Technology

Our backend consists of five different deployable units, each developed using Django to perform the business logic, data logic, and API functionality. The reasoning behind this technology choice is explained below.

6.2.1 Django

The following reasons are the main driving force this design decision:

- Django is, by design, highly secure 'out of the box' and provides comprehensive protection for all the most common security threats (SQL injection, CSRF, clickjacking, etc). This is of particularly high value to us since (as pointed out in the architectural constraints) the system will be hosted on a virtual machine with a public IP that is subject to frequent cyber-attacks. In order to minimize risk wrought on by inexperience or oversight, a highly secure framework such as Django is a very suitable choice for our purposes.

- Django is an entirely Python development environment. Since our system includes a significant portion of data manipulation (across all services), Python will improve the ease with which we are able to develop the system. Furthermore, Python has a number of fantastic libraries for Natural Language Processing and Machine Learning (ex: NLTK, PyTorch, etc), this language is a great choice to handle the application's sentiment analysis component. Additionally, all group members have high proficiency with Python, allowing us to lean into our strengths as programmers.

- Django is designed for rapid and smooth development, this is suitable and desirable since we are adopting the Agile Software Process. That is, by developing our system using Django, we are able to rapidly prototype and build on different aspects of our system - allowing for maximum flexibility, while speeding up iteration-by-iteration improvement (highly desirable and necessary trait to successfully adopt the Agile process)

However, no technology choice is ever perfect and there are some trade-offs which need to be minimized.

- Django accomplishes the advantages pointed out by points 1 and 3 above by being a 'batteries included' framework. Unfortunately this means that Django is more heavyweight than other Python frameworks such as Flask. A more lightweight framework may be more desirable since we are adopting a service-oriented architecture (ie: not every service will have use of all of Django's built-in features). We do however believe that the advantages described above outweigh this point, since each service in the architecture will leverage at least a few of Django's native features to great effect.

- Django is somewhat opinionated about the structure of the application which once again is not particularly desirable for a service-oriented architecture - however, this point is easily negated as it is entirely possible (and fairly easy) to add (through the use of user-defined Python modules) and remove components to accomplish our architectural structure.

6.3 Database Technology

Our system comprises three seperate databases, each running on different deployable units and each serve a different purpose. The description and reasoning of each is below.

6.3.1 Sentiment Records Database

This is a NoSQL document-based MongoDB database that runs on the Data Warehouse deployable unit. It is the main and largest database on the application, and contains all the sentiment records/data that have been analyzed, as well as the sentiment metrics that have been computed for each record.

- A NoSQL database lends itself to this data because of unpredictable size and variety of the sentiment records that are retrieved online.
- Furthermore, the metrics that are computed for each record are closely tied to the record itself (since the record will never be accessed outside the context of checking the sentiment metrics). By leveraging the document structure allowed by MongoDB, it is possible to store both the data and the metrics as a single document/unit.
- Potentially most importantly, a NoSQL database (especially the document-based MongoDB) allows for flexibility and modifiability of the metrics computed on a particular sentiment record. For example, in the future it may be desirable to include an objectivity-subjectivity score on new articles that have been analysed. The flexibility of MongoDB would make this modification extremely simple, as no change needs to be applied to the database schema, or to any of the other existing documents in the database.

6.3.2 Domains and Sources Database

This is a NoSQL document-based MongoDB database that runs on the Domains deployable unit. It is responsible for storing the information and structure of a domain, as well as the sources specified within it (including the information and parameters for those sources)

- Since domains effectively act as 'folders' for sources, MongoDB is once again a suitable choice of technology, as it allows us to store sources as nested objects within a domain (which is the document). This helps accomplish 1. domains physically group their associated sources (and easily catering for a variable number of sources), as opposed to a SQL approach, which would alomost certainly require the use of complex joins, and 2. accomplish some form of query/database optimization in that since domains will ALWAYS be accessed in order to retrieve ALL their contained sources, we avoid consistently having to use joins like we would in a SQL databasee, instead by retrieving a single document (domain) we have all the information we required.
- Furthermore since the parameters for different sources vary both in number and in type/name, by leverage a MongoDB database we do not need to create some static schema that needs to cater for all possible parameters. Instead we can simple create an object that contains key value pairs of the parameter name and its value.

6.3.3 Profiles, Users, and Authentication

This is a SQL Postgre database that runs on the Profiles deployable unit. It is resposible for storing a user's profile information and preferences, linking it to their domains in the above database, and managing user and authentication data.

- The data to be stored by this database is highly structured and well-defined, and consequently a SQL database is highly suitable for this purpose.
- Additionally, Django (backend technology) provides seemless and secure integration with Postgre 'right out of the box' and hence we are able to leverage this predefined interaction in our development. (It is worth noting that this seemless integration is designed with security in mind, providing appropriate and comprehensive protection for database attacks such as SQL Injection)