Application of web design techniques and best practices in implementing web development, maintenance and enhancement of RUBi websites and web application systems

Rhodes University

Makhanda, Eastern Cape, South Africa

Professor Ozlem Tastan Bishop

by

Thulani Tshabalalala

Research Unit in Bioinformatics, Department of Biochemistry and Microbiology

Abstract

The popularity of the web has seen various fields, such as the sciences taking advantage of this resource to further their scientific endeavours. This has seen science groups moving into developing websites and web applications, and such a group is the Research Unit in Bioinformative (RUBi). With the use of the web, the development and maintenance of whatever web-related tools become inevitable, given the continuous changes in the webspace. This continuous evolution of web development and maintenance will come with techniques, principles and standards which will not only enable faster development of web entities but also ensure that modern hardware, fulfilment of the requirements to use such hardware and modern concepts are incorporated into forming web tools that enable such progression. Furthermore, introducing the previously mentioned progress of the web becomes an essential part of its development and maintenance. This paper did implement the processes of progressing the web using the technique of documentation and version control systems. The web development for the COVIDRUG website was done for the Covidrug-Africa Consortium (COVIDRUG) using the Django web development framework. The RUBi website and the MDM-Task web and the Job Management System (JMS) web applications were maintained for the maintenance aspect. Archives brought value regarding the traceability it provides of the various web-related aspects. The development showed a website's potential value, particularly for research groups. The maintenance carried out showed how different techniques and approaches could be used in different maintenance prospects to achieve set objectives. The development and maintenance resulted in websites and web applications that have the features stated in their respective maintenance plans.

Contents

List of Figures

List of Tables

Acknowledgements

First, I would like to express my appreciation and gratitude for Lithalethu Hashe, who encouraged and supported me throughout this process. To my family, friends and colleagues, I thank you for the laughs and happy moments that helped me endure difficult times.

Most importantly, to the funders of the Research Unit in Bioinformatics (RUBi), namely: National Human Genome Research, Institute of the National Institutes of Health under award number U24HG006941 to H3ABioNet and U3ABioNet6941 to H3ABioNet for providing the funding resources, so RUBi under the guidance of Professor Ozlem can continue its fantastic work and contributions to the field of Bioinformatics. For this, I would like to thank Professor Ozlem for giving me the opportunity to be part of RUBi, where I gained knowledge, skill sets and experience that I will forever hold valuable.

Overview

Web development and maintenance have different aspects, both relating to the hardware and software that enables the web interactions between websites and web clients. For this reason, it is important to understand both these aspects of the web. However, the discussion that will be had in this paper will be with regards to the software aspect of the web with a brief overview of the hardware aspect.

For the software aspect of web development and maintenance to be had, an introduction is given about the different aspects that relate to the software and attributes aspect involved in the web. This will contain the different software tools that can be used to build and host web applications. Followed by an in-depth discussion regarding the Django web development framework, which will be the framework that is used for the development and maintenance objectives that need to be fulfilled. Once the construction of a website or web application is understood, the resources used in hosting these websites or web applications is an important part of understanding how web clients gain access.

However, before the task of carrying out any web development or maintenance, a traceability process had to be fulfilled. This process involves the documenting of the *Jabba* storage server and the use of a version control system (VCS) to create repositories to archive the software tools on the *Padme* server. Furthermore, these repositories were used throughout the development and maintenance process.

Once this was done, the development of the COVIDRUG website was done, which is used by the Covidrug-Africa Consortium. The website gives various information regarding the consortium as it aims to contribute to the response against the Coronavirus Disease. Coupled with this development was the maintenance of the Research Unit in Bioinformatics (RUBi) website, which provides interested web clients with information regarding the research unit.

Lastly, two web applications were maintained, namely MDM-Task Web and Job Management System (JMS). The maintenance carried out on MDM-Task Web adds new functionality and expands existing functionality. This also included various additions that enhanced the construction of the MDM-Task Web in the environment in which it is hosted. The maintenance of JMS was updating the version of the back-end programming language and replacement of the job scheduler used by JMS.

All of this showed how web development and maintenance could be implemented in different websites and web applications, including the techniques which were developed and implemented to achieve the objectives set out in this discussion.

Chapter 1

Literature Review

The World Wide Web (WWW) continues to provide a repository of resources whereby various web clients that have access to the web via the internet can access web servers containing different information and functionalities (Wang, Huang, Qu, & Xie, 2004; Foster, Kesselman, & Tuecke, 2001). Historically, the web primarily gave web clients access to web servers that host websites that contained simple text and images via request protocols (Casal, 2005; Foster et al., 2001). This static nature of websites evolved to be more dynamic, which gives web clients a more interactive experience (figure 1.1).

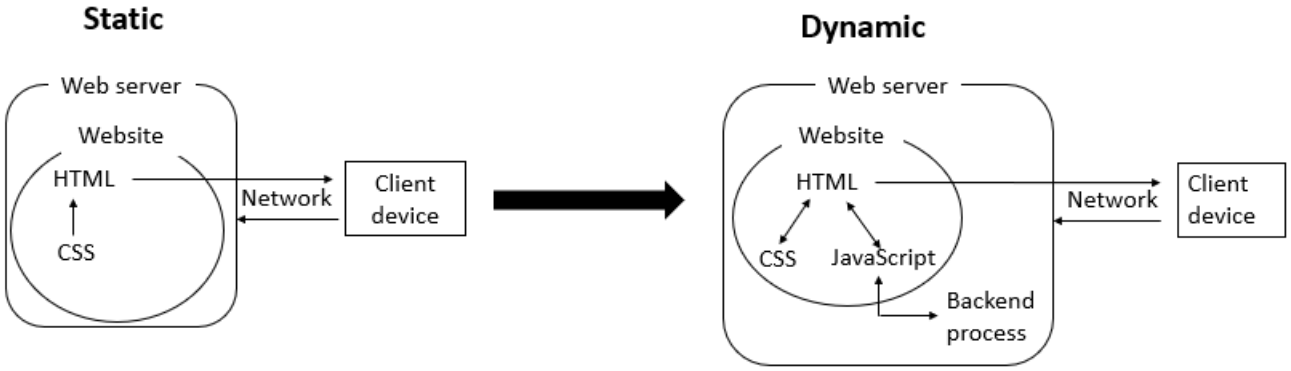

Figure 1.1: Evolution of static to dynamic websites and the components involved

The evolution of websites from static to dynamic was achieved by adding components such as JavaScript to enable interactivity, while the back-end process can provide additional functionality.

These static websites are built using Hypertext Markup Language (HTML) and Cascading Style Sheets (CSS). Although HTML tags such as buttons can result in website interactivity, this interactivity does not manifest within a web page but rather in forms such as hyperlinks to other web pages. Therefore, these HTML tags remain limited in their potential. Dynamic websites are made possible by using primarily JavaScript, and via JavaScript, a back-end process can be added to enhance the functionality of that given website.

The evolution of website introduced new tools which can be used to facilitate the evolution from being static to being dynamic (figure 1.1). The first step in moving the web in a dynamic framework, was the development of JavaScript (Sun & Ryu, 2017). JavaScript is a scripting programming language which can generate functionality during run-time and can also be embedded into HTML (Sun & Ryu, 2017). These two features enable the enhances of HTML tags such as buttons amongst others to trigger various changes to the current web page (Sun & Ryu, 2017). Furthermore, JavaScript can also provide computational and logic functionality to the user or system website input via HTML tags (Sun & Ryu, 2017). Therefore JavaScript enables websites to be interactive and responsive thus the dynamic categorisation (Sun & Ryu, 2017).

The responsive attributes of website design has enabled new attributes to now be developed and enabled the creation of web applications (Yousaf, Arshad, Nouman, & Arshad, 2018). Web applications are applications that exist within the browser environment and carry a certain functionality which the web client can have access (Jayamsakthi Shanmugam, 2008; Casal, 2005). Furthermore, web applications can be categorised into static web applications, and dynamic web applications (Jayamsakthi Shanmugam, 2008). Both of these broad categories are interactive, whereby the interactive nature of static web applications are done by front end components like HTML, CSS and JavaScript (Y.-F. Li, Das, & Dowe, 2014). On the contrary dynamic web applications are typically associated with server side attributes such as databases that would work hand in hand with the front end components previously mentioned (figure 1.2).

Figure 1.2, shows the different ways in which are web applications can be designed and the components that are required to develop that design. Due to the diverse usage of web application in modern websites, web applications constitute a larger portion of the web as the web becomes more accessible to more people (X. Li & Xue, 2014). The growth of the usage of web applications and their accessibility has raised various concerns and challenges (Y.-F. Li et al., 2014; Jayamsakthi Shanmugam, 2008). Due to web applications being exposed to any web client on the web and some being connected to a database and other server side (back end)

related software. The need to maintain of security and control access to those different parts begins to be increasingly important (Gnanam, 2016; Y.-F. Li et al., 2014). There are different aspects that, when applied, can dictate how maintenance and access are controlled in the short and long term (Gnanam, 2016; Y.-F. Li et al., 2014).

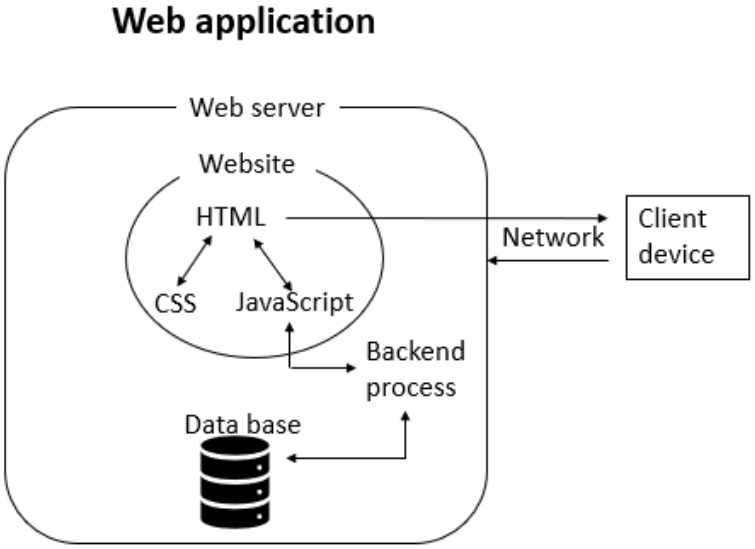

Figure 1.2: Abstract view of a web application and components

The different components that constitute static web applications and dynamic web applications. The differentiating component being the connection of a database to dynamic web applications.

Web applications face risks from different locations as previously mentioned (Gnanam, 2016; Deepa & Thilagam, 2016; Y.-F. Li et al., 2014). The risk can come from front end components to back end components (Gnanam, 2016; Deepa & Thilagam, 2016). Therefore, the implementation approaches used to develop the different components of the web application can either be prone to particular risks or minimise those risks. So there is a need to be mindful of these risks and their nature by ensuring that the steps in the implementation approach are robust to minimise those risks (Gnanam, 2016; Deepa & Thilagam, 2016). There are vast implementation approaches that can be derived from the predominant web development approaches, namely Adaptive Web Design (AWD) and Responsive Web Design (RWD) (Yousaf, Butt, Azam, & Anwar, 2018). Different devices such as desktops, mobile phones or laptops can access the web as web clients. These devices have different attributes, from system components to operating systems. Therefore the AWD approach would be more oriented in ensuring that the web application in question adapts its design given different device environments (Yousaf, Butt, et al., 2018). On the other hand, the RWD approach would ensure that the web application carries

the same responsiveness in different device environments (Yousaf, Butt, et al., 2018). These two predominate web design approaches and their derivatives are prone to different risks that can be mediated during a robust mindful implementation approach.

However, various factors affect the implementation approach regardless of its robustness or mindfulness. Some of these factors are deployment deadline, human resources, available tools, development practices etc (Prokhorenko, Choo, & Ashman, 2016; X. Li & Xue, 2014). Deployment deadlines affect general oversight on whether the implementation approach has been fully implemented (Prokhorenko et al., 2016; X. Li & Xue, 2014). Failure of oversight tends to result in issues such as poor testing of the risks that the web application is susceptible to due to the relatively short deployment deadline (Prokhorenko et al., 2016; X. Li & Xue, 2014). This may also indicate that not enough time and understanding was spent during the implementation planning to ensure the deadline set will also allow for the necessary steps to be implemented while still allowing time for the unknowns (Prokhorenko et al., 2016; X. Li & Xue, 2014). Furthermore, human resources exaggerate the effects of deployment deadlines due to the availability of people to successfully implement the selected approach (Prokhorenko et al., 2016; X. Li & Xue, 2014). Available tools speaks to the resources made available to ensure the selected implementation approach can be delivered on the set targets (Prokhorenko et al., 2016; X. Li & Xue, 2014). Various tools like Integrated Development Environments can offer functionalities like debugging, which can enable developers to test whether the components in the implementation approach selected are function securely (Zhevaho, 2021; X. Li & Xue, 2014). Such tools also contribute to development practices which would constitute what techniques are used to ensure that the web application implementation process is robust (X. Li & Xue, 2014).

Development practices can determine the web application usability, and life cycle, the reason being that particular implementation designs and protocols ensure continuous web applications usability and a longer life cycle (Prokhorenko et al., 2016; X. Li & Xue, 2014). Front end implementation designs that have validation procedures form part of development practices that improve the usability of a web application (Deepa & Thilagam, 2016; X. Li & Xue, 2014). Validation procedures ensure that a web application does not suffer from front end malicious activity, which can cripple a web application (X. Li & Xue, 2014). Such validation procedures can be used by checking whether the type of interaction web clients are having with the front components are in line with the functionality offered by the web application (X. Li & Xue, 2014). Examples being type or pattern checking which ensure the inputs from the web client

are within the expected type or pattern (figure 1.3).

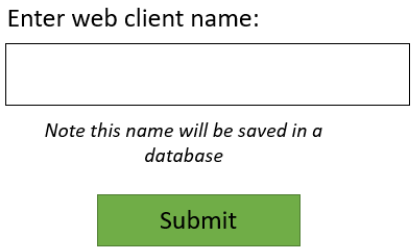

Figure 1.3: Web application components for storing web client use name

Web application components that requires the web client to enter their user name that will then be saved to a database after the web client clicks on the submit button.

Figure 1.3, shows a web application component that is prone to SQL injection risk (Lawal, Sultan, & Shakiru, 2016). This type of risk can be mediated by using type or pattern checking validation procedure mentioned previously in the case of the functionality offered by these web components (figure 1.3). The validation procedure would ensure that web clients do not attend to compromise the web application using Structured Query Language (SQL) injection techniques (Lawal et al., 2016). SQL injection is the procedure of writing SQL commands in web client input fields of a database connected web application that results in those commands being executed (Lawal et al., 2016). Therefore type or pattern checking validation procedure can ensure that web clients do not use the user name input field in the instance of figure 1.3, for SQL injection.

However such protection procedure as validation web client input are implemented as a code level and can only mediate particular risks. Software applications like antivirus can also be used to carry out such protective functions by outsourcing the function of protection from the web application front end code to the software tools (Prokhorenko et al., 2016; Deepa & Thilagam, 2016; X. Li & Xue, 2014). Lastly, procedures like access control in web applications can form a protective layer from malicious activity that may affect back end components like databases ensuring all the web clients that have access have have been sanitised (Deepa & Thilagam, 2016; X. Li & Xue, 2014). This can be implemented by using a user registration fashion, predetermined Internet Protocol (IP) address being granted access to device identification meta data to grant access to particular devices etc (Deepa & Thilagam, 2016; X. Li & Xue, 2014). The previously mentioned are just a few protective techniques that can be considered during the implementation process to ensure the usability and thus the life cycle of the web application is lengthened.

Testing of the web application once various procedures have been implemented is vital (Deepa & Thilagam, 2016; X. Li & Xue, 2014). The testing of web applications during the development phase enables the discovery of new vulnerabilities which need to be patched (Deepa & Thilagam, 2016; X. Li & Xue, 2014). Additionally, the testing can enable the developers to observe the web application performance under various conditions, determine whether improvements are required and whether the intended functionality of the web application has been met using the selected implementation approach (Deepa & Thilagam, 2016; X. Li & Xue, 2014). However, a comprehensive testing procedure can still not determine all the bugs and vulnerabilities contained in the web application (Deepa & Thilagam, 2016; X. Li & Xue, 2014). Therefore techniques like partial deployments can be added into the testing phase to determine possible vulnerabilities that were not discovered (Deepa & Thilagam, 2016; X. Li & Xue, 2014). Partial deployment gives web clients who are not part of the implementation team access to the web application to use and run various tests, thus giving the web application a set of new eyes to seek possible issues (figure 1.4).

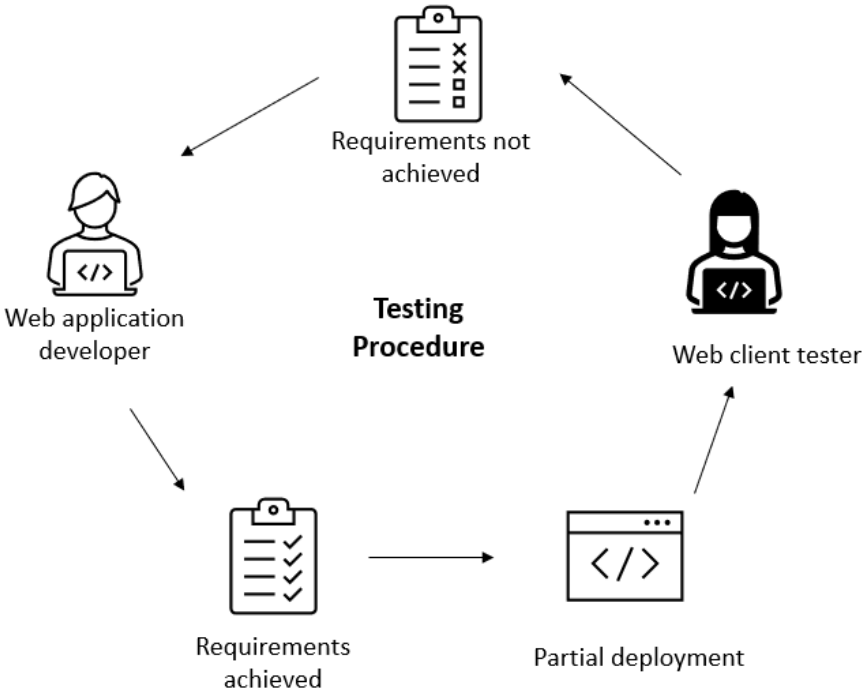

Figure 1.4: Web application testing procedure

The testing procedure that can be used to ensure the functionality that is carried out by the web application has been implemented successfully.

Figure 1.4, shows how the technique of partial deployment can be used in the testing of web applications. The cycle of the partial deployment testing procedure is only broken once the web clients who are doing the testing are satisfied that the requirements have been met (figure 1.4). Furthermore, web applications can be deployed in an environment that contains tools that enhance the functionality and safety of the various components of the web application (Deepa & Thilagam, 2016; X. Li & Xue, 2014). These enhancement tools can have threat detection, and prevention features (Deepa & Thilagam, 2016; X. Li & Xue, 2014). Threat detection features can inform the developers of such a web application if possible malicious activity happens, which can compromise the web application infrastructure. In contrast, preventive features can stop the web clients that are carrying out the malicious activity (Deepa & Thilagam, 2016; X. Li & Xue, 2014). The previously mentioned enhancement tools should always be incorporated in the implementation of any given web application to ensure that code level measures are also complemented by proactive measures that have been mentioned.

Due to the continuously changing environment in the web space, whether those changes are browser related, network related or in developmental tools, these changes affect the life cycle of a web application and the best practices which are involved in selecting the implementation approach (Deepa & Thilagam, 2016; Prokhorenko et al., 2016). Primarily the changes mentioned previously are driven not only by the need to improve the usability of the web content and thus web applications but also as a response to threats that make older web technologies vulnerable (Deepa & Thilagam, 2016; Prokhorenko et al., 2016; X. Li & Xue, 2014). This changing environment also requires the implementation approach selected to also result in a web application that is maintainable (figure 1.5).

Figure 1.5, shows how the maintainability of a web application will determine the life cycle of the web application (Tian et al., 2021; Mohan & Greer, 2018). Furthermore, the maintainability of a web application will indicate how many changes the web application can absorb while maintaining its primary function (Tian et al., 2021; Mohan & Greer, 2018). Maintainability is achieved by how the source code of the different components of the web application are constructed (Tian et al., 2021; Mohan & Greer, 2018). If a web application is not maintainable, any changes that require additions to the web application tend to result in the web application being decommissioned (Tian et al., 2021; Mohan & Greer, 2018). Furthermore, the maintainability of a web application, as previously mentioned, also affects the security prospects of the web application (Tian et al., 2021; Mohan & Greer, 2018).

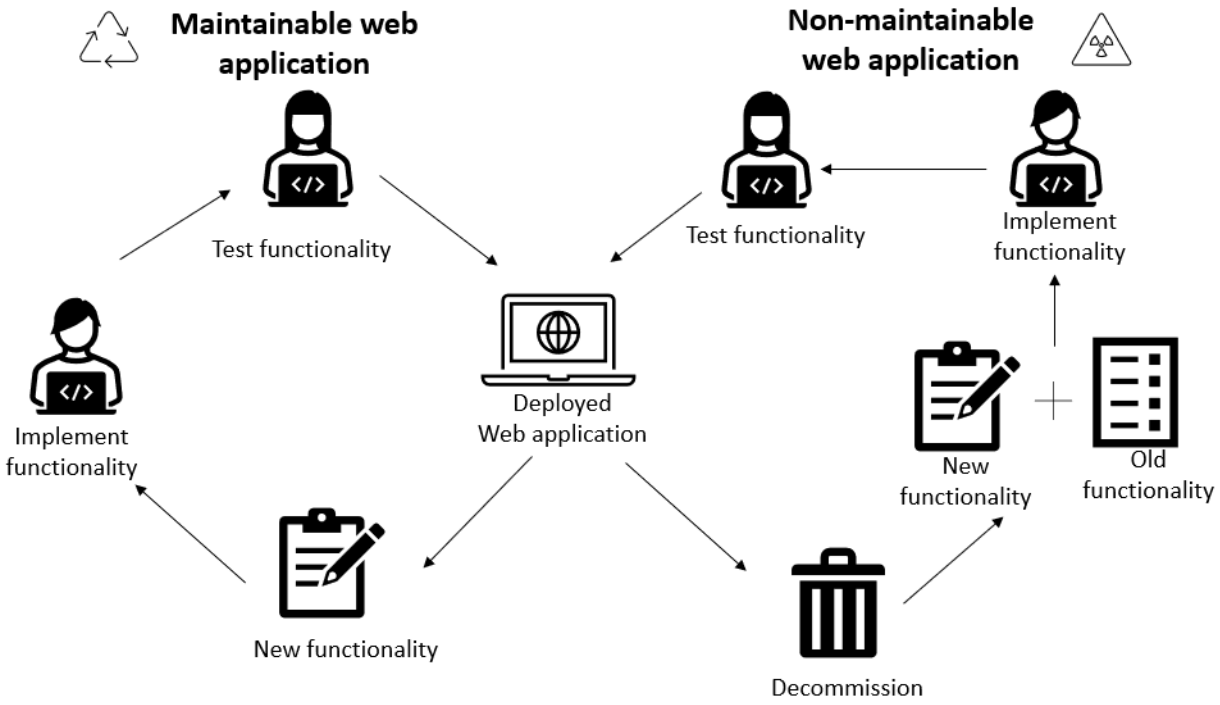

Figure 1.5: Addition of new functionality to a maintainable and non-maintainable web application

The process that is involved in adding new functionality to a web application based on its maintainability attribute.

Due to new techniques being continuously developed to counter and prevent security breaches, the implementation of these new techniques will require a web application that is also maintainable (Tian et al., 2021; Mohan & Greer, 2018). Therefore maintainability becomes a primary factor when considering the usability and life cycle of web application construction (Tian et al., 2021; Mohan & Greer, 2018).

There are various techniques in which maintainability can be ensured during the construction of a web application (Tian et al., 2021; Mohan & Greer, 2018). These techniques can include ensuring that the various components of the web application, for instance, the front end components like inputs objects, are loosely coupled (Tian et al., 2021; Mohan & Greer, 2018). This type of loosely coupled components technique enables the addition and removal of features from the web application without affecting the functionality of other components in the web application (Tian et al., 2021; Mohan & Greer, 2018).

However, continued maintenance can result in web application component coupling, particularly in a rapidly changing environment that requires multiple additions to be made to the web application, particularly to legacy web applications in deployment (Tian et al., 2021; Mohan & Greer, 2018). This can be a result of compatibility issues of the additions that are required, to structural limitations (Tian et al., 2021; Mohan & Greer, 2018). Although a web application can be maintainable, if its technology does not comply with the required additions or improvements, this too can result in decommission (Tian et al., 2021; Mohan & Greer, 2018). Therefore, to ensure that a web application's maintainability bears fruit, various considerations need to be made during the construction of the web application (Tian et al., 2021; Mohan & Greer, 2018).

The considerations that need to be made can include the long term viability of the technologies being used, whether the components being used can support such viability and how coupled is the logic of the web application to particular technologies (Tian et al., 2021; Mohan & Greer, 2018). It is also worth noting that paying consistent attention to the developments occurring in the technologies that being used by the web application is important. This is to ensure that issues resulting from those development can be identified early so appropriate measures can be taken (Tian et al., 2021; Mohan & Greer, 2018). Once a suitable outcome regarding the technologies that will be used in the implementation has been reached, it is essential to document the developmental process of the web application. Such documentation can ensure that when maintenance is required, traceability can be applied (Tian et al., 2021).

Traceability is achieved by the process of documenting all the functionality of a given software and the techniques used to implement the functionality (Tian et al., 2021). Traceability by the usage of documentation can inform the developers how particular additions or changes will affect the web application in this instance, and to ensure that the maintenance is implemented with minimal structural compromise (Tian et al., 2021). Traceability of any software platform has the potential to increase productivity in regards to making improvements and additions. However, factors previously mentioned in this chapter also affects traceability (Tian et al., 2021).

The continuing improvements in the web space also brings with it web development frameworks that gives web developers different advantages based on the type of web development required (del Pilar Salas-Zárate et al., 2015). Figure 1.6, shows under the Web application development tools header how the tools are used in a typical construction of a web application. All the different tools have to be brought together via the shown interactions to carry out the functionality of the web applications (figure 1.6). The Web development frameworks header shows how a framework contains all the necessary tools for web development in a single environment, where the interactions between the tools are defined by the framework environment (figure 1.6).

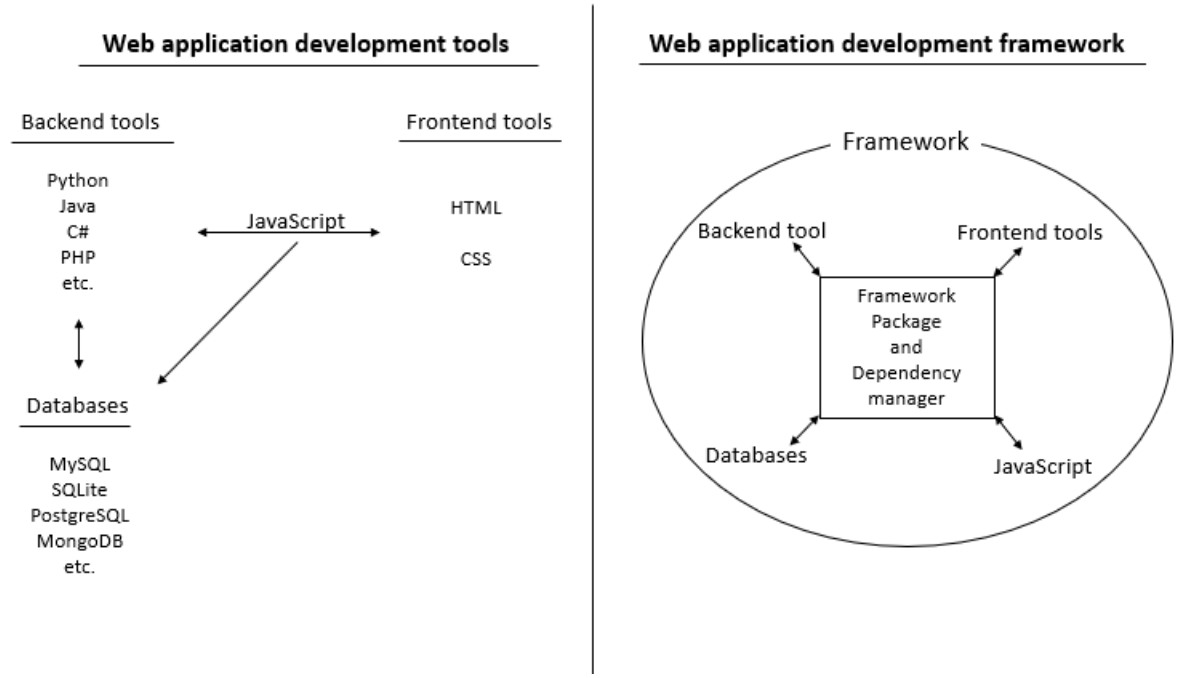

Figure 1.6: Web development components tools and the relation to web development frameworks

The individual tools that can be used in the development of a web application on the left and how web development frameworks bundles these tools together in their environments on the right.

There are various web development frameworks such as CakePHP, Node.js, Ruby on Rails, JavaServer Faces (JSF) and Django to mention a few (del Pilar Salas-Zárate et al., 2015). These web development frameworks offer developers resources and library tools that make some aspects of web development productive (del Pilar Salas-Zárate et al., 2015). These resources and library tools tend to contain functionalities such as front end components like buttons that have all the necessary attributes, enabling developers to call these resources or library tools without needing to construct the component (del Pilar Salas-Zárate et al., 2015). Additionally, some of these web frameworks enable full stack development of a web application (del Pilar Salas-Zárate et al., 2015).

However, as previously mentioned, without an appropriate implementation approach that has all the necessary techniques for the requirements of the web application, the selection of a framework can be futile (del Pilar Salas-Zárate et al., 2015). In regards to frameworks, they tend to be based on a programming language, particularly the back end programming language, since the standard front end is designed using HTML, CSS and JavaScript (del Pilar Salas-Zárate et al., 2015). This is important since an inclusive implementation approach would consider which programming language the developers are most competent and efficient to ensure speedy implementation (del Pilar Salas-Zárate et al., 2015). These various tools give developers the ability to promptly and securely develop and maintain web applications. Although there are multiple challenges and limitations given the availability of resources during development or maintenance, such challenges and limitations can be minimised with a conscious implementation approach.

Figure 1.7, shows the first iteration of how traditional scientific work was transformed in order to take advantage of the increasing popularity and accessibility of the web. However, this modernisation of scientific work is biased toward particular types of scientific work. The reason is that the purpose of some scientific work is not to produce knowledge about observations, but rather tools to carry out those observations (Abriata, 2017; Chen, Yu, & Chen, 2013). The bias is that the tools perform logic, computation and analysis for a given observation. Therefore this type of scientific work requires a new system that is not catered for in the electric copy sphere about those given tools since other interested parties may want access to those tools (Abriata, 2017; Chen et al., 2013). The increasing popularity of web application mentioned previous patches this issue (Abriata, 2017; Chen et al., 2013). Historically, these tools were done as programs that meant they were localised to a given machine. However, when these programs are converted to a web application using the technologies previously mentioned, this

type of scientific work can also take advantage of the web as well. This type of web application design enabled traditional biological science practices to expand to a larger group of researchers, making collaboration efforts more convenient (Abriata, 2017; Chen et al., 2013).

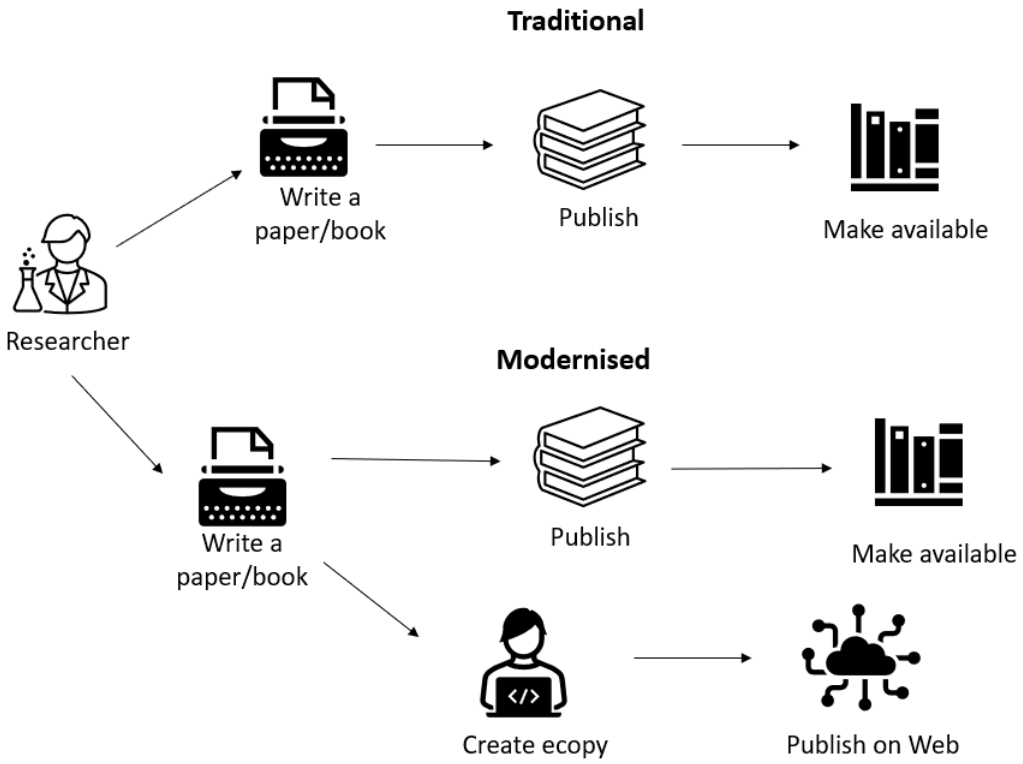

Figure 1.7: Overview of the difference between the traditional and modernised flow of scientific work

How the development of the web and its distributive nature enabled scientific to be shared while maintaining more traditional means of knowledge sharing.

With this in mind, it can be argued that the biggest potential which the web has offered biological sciences are analytical and descriptive web applications that web clients can access (Abriata, 2017; Chen et al., 2013). These analytical and descriptive web applications offer web clients the ability to do their research and derive credible results from these web applications (Abriata, 2017; Chen et al., 2013). Bioinformatics as a biological sciences field has experienced the most significant yield from such web applications, given that these web applications can be configured to support various parts of bioinformatics from visualisation to computation to storage (Abriata, 2017). There are multiple requirements for an analytical and descriptive web application to be successfully implemented and hosted.

1.1 Frameworks

As mentioned in the introduction, various web development frameworks can be used for any web application development, including biological science web applications. A framework like CakePHP, which is a scripting language that supports the development and maintenance of web applications that use Hypertext Preprocessor (PHP) (Reniers, Van Landuyt, Rafique, & Joosen, 2019; Birnbaum, 2010). PHP is a scripting language that was designed for being executed on the server-side, making it suitable for web design due to websites are typically hosted on servers (Reniers et al., 2019; Birnbaum, 2010). PHP became one of the first programming languages that gave the web its interactive ability giving a framework like CakePHP an advantage due to PHP widespread usage in web design (Reniers et al., 2019; Birnbaum, 2010).

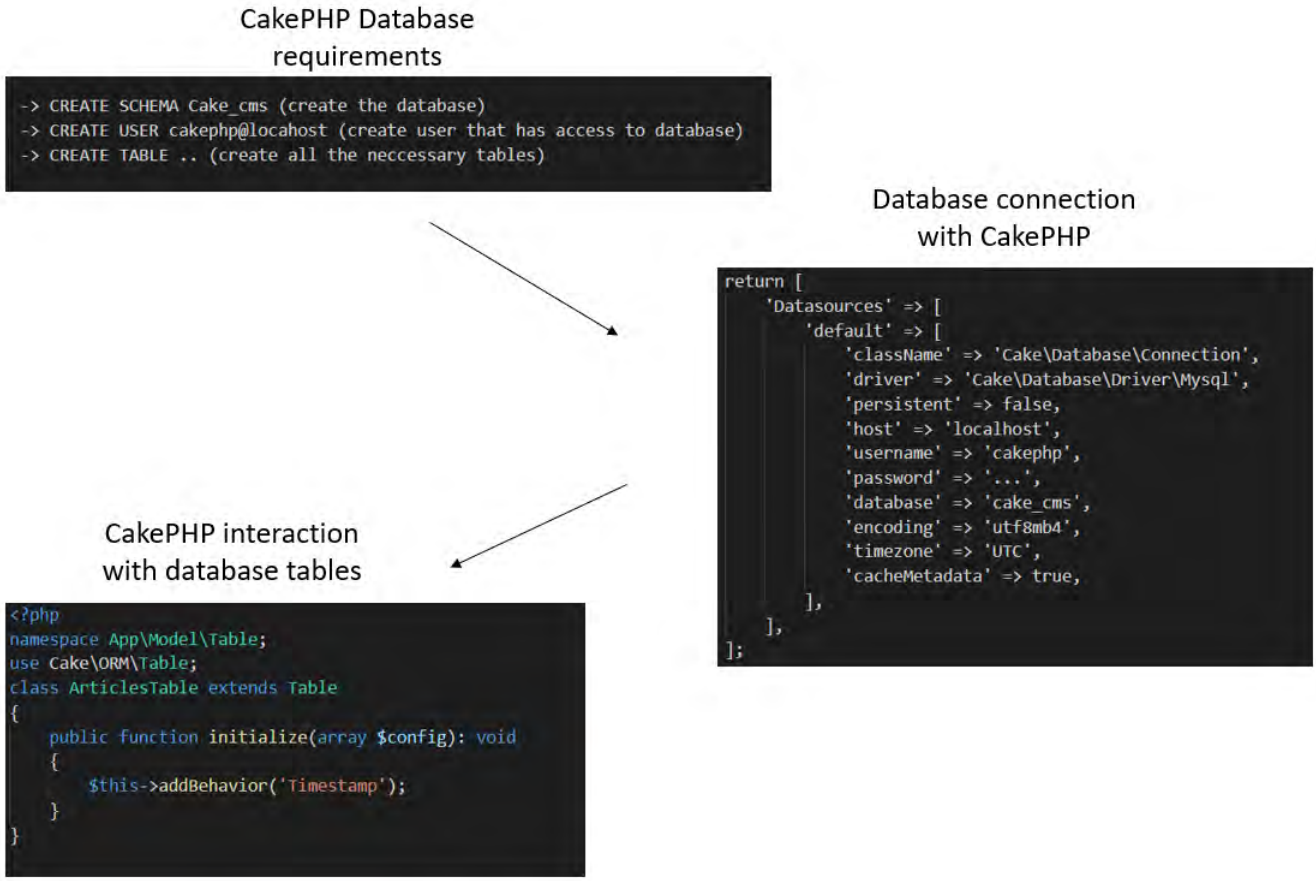

Figure 1.8: Process of enabling database interaction in CakePHP framework

The process used to create the database that will be accessed by a given CakePHP designed web application. This is followed by the user name that the web application will use to write to the given database. Then lastly, the tables that will be used in the web application. The details regarding the database need to be provided to the CakePHP web application, which can then interact with the tables in the given database.

The CakePHP framework does not have internal database creation procedures (Reniers et al., 2019; Birnbaum, 2010). Therefore for the framework to be connected to a database, the developer will need to create a database that has all the required parameters for the CakePHP framework database connection (figure 1.8). Figure 1.8, shows that in order to create a CakePHP environment that has all the needed tools for the development of a web application, an external database creation step is necessary. In this instance, a MySQL database is used, but CakePHP does provide support for various relational database software (figure 1.8). Lately, this type of interaction with a database tends to give the framework in question protection against SQL injections by using a different query pattern procedure in its database interaction (Birnbaum, 2010).

Node.js framework is a collection of different frameworks that share commonality in their usages of JavaScript in their full stack web application developments. This means that Node enables the execution of JavaScript outside the browser environment which typically only exist in the browser environment. An example of a Node framework is Meteor, that can be used to develop both web application and mobile application (Meteor contributors, 2022). The usage of JavaScript in backend application entails that Node frameworks such as Meteor need to be environment oriented (Meteor contributors, 2022). This environment orientation means that the Meteor framework needs to have functionalities that inform the web application in this instance of whether the functionality being executed is on the frontend or backend (Meteor contributors, 2022). The reason being that some functionalities execution need to remain in the backend while other functionalities can be executed at in any of the two environments (Meteor contributors, 2022). Such functionalities that may need to remain in the backend are resource requiring functions that need the resource capacity of the server such as processing power to complete their functionality (Meteor contributors, 2022). It is important to ensure that these such functions remain in the backend (server) environment, since the browser may not have sufficient resources if such functions were executed in the frontend (Meteor contributors, 2022). Figure 1.9, shows the mechanisms the Meteor framework uses to determine the execution environment.

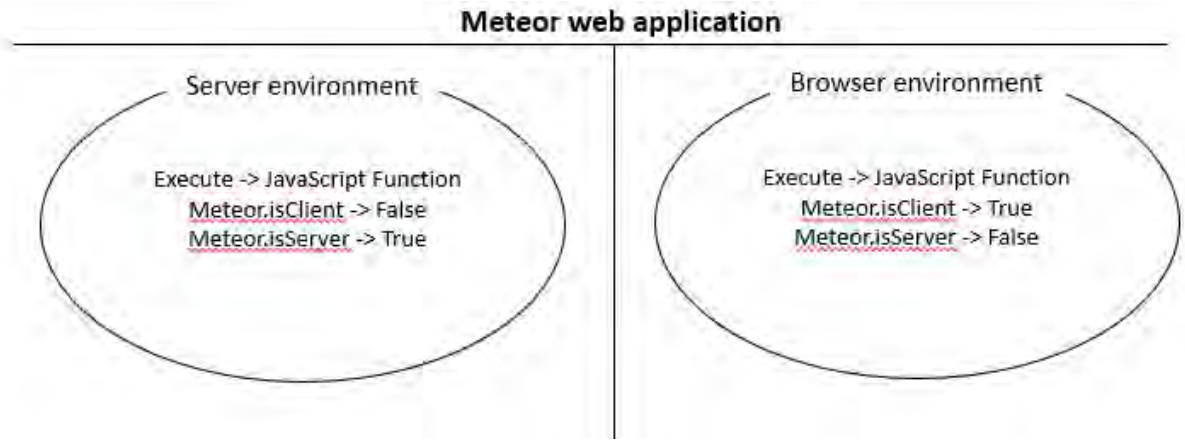

Figure 1.9: Meteor web development framework mechanism for environment detection

The *.isClient* and *.isServer* boolean functions, that are used by Meteor web development framework to inform the developer during run time of which environment the given function is being executed.

The usage of the *.isClient* and *.isServer* boolean functions in Meteor can enable the developer to inform the web application about which function to execute in which environment given the requirements of that particular function (Meteor contributors, 2022). However, it is worth noting that the Meteor web development framework also provides a directory system then also informs where the function is executed based on its directory (Meteor contributors, 2022). The reason for this mechanism is that the *.isClient* and *.isSever* boolean functions only prevent the function from being executed but the function will be present in both environments (Meteor contributors, 2022). This can pose security issues as the presence of functions such as password storage and authorisation functions in the frontend environment leaves a possibility of those functions being accessed by malicious web clients (Meteor contributors, 2022). Therefore, the directory system can ensure that sensitive functions previously mentioned are not sent to the frontend environment (Meteor contributors, 2022).

Ruby on Rails web development framework was developed to use the Ruby programming language (Ruby Rails contributors, 2022). This framework also uses the Node.js framework and Yarn package manager (Ruby Rails contributors, 2022). These two software tools enable the Ruby on Rails framework to manage the JavaScript aspects (Ruby Rails contributors, 2022). Furthermore, the framework requires SQLite3 database software (Ruby Rails contributors, 2022). However, the Ruby on Rails framework can use other relational database software (Ruby Rails contributors, 2022).

The primary advantage of a web development framework such as Ruby on Rails is that it comes with generator functions that are programmed to generate various scripts that are necessary to start the developmental process (Ruby Rails contributors, 2022). Figure 1.10, shows the execution pattern of the Ruby on Rails framework. This is how a web application built using the Ruby on Rails framework generates its interaction between its three main components for the web application functionality (figure 1.10).

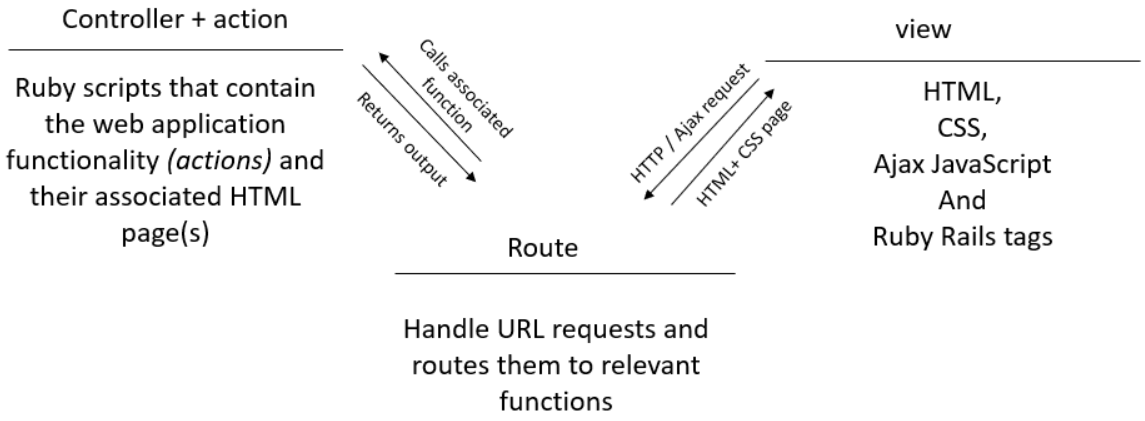

Figure 1.10: Ruby on Rails web development framework execution pattern between its components

The components that enable a Ruby on Rails web application to carry out its functionality. The *view* will contain all the contents that the web client sees and interacts with. The *route* component connects the requests the web clients send to the Ruby functions that service those request using the *controller* and *action* concepts of Ruby on Rails.

JavaServer Faces (JSF) is a framework that uses the Java programming language for developing its back-end functionality to build web applications that can be classified as user interfaces (Jurisic & Kermek, 2014). The concept of using user interfaces enables web developers to use this framework to develop their web applications in the context of components (Jurisic & Kermek, 2014). These components enable decoupling by having functionality existing in its compartment while improving customisation by developing functionality that is specific to that particular component (Jurisic & Kermek, 2014). JSF uses the Model-View-Controller (MVC) methodology, which enables the development implementation to control the interaction between the front-end (view) and the back-end (model) using the controller concept (Jurisic & Kermek, 2014). Figure 1.11, shows JSF deployment of the MVC methodology in a web development instance.

The JSF web development framework, therefore, enables developers with different skill sets to participate in web development and in this instance, these would be frontend developers and Java backend developers (figure 1.11). In this instance the Java convention of using *setter* and *getter* functions to enable interaction between the view and model is used for information flow via the controller (figure 1.11). The HTML document will have JSF tags which are triggered when an event is passed onto those embedded JSF tags (Jurisic & Kermek, 2014). The *getter* and *setter* functions associated with those JFS tags will then get the functionality required for that event from the Java backend and set that functionality onto the frontend (Jurisic & Kermek, 2014). As previously mentioned, such a design in a web application does not only produce flexibility, but the decoupling between the frontend design and the backend logic, enabling the addition of new logic and maintenance of the web application easier (Jurisic & Kermek, 2014).

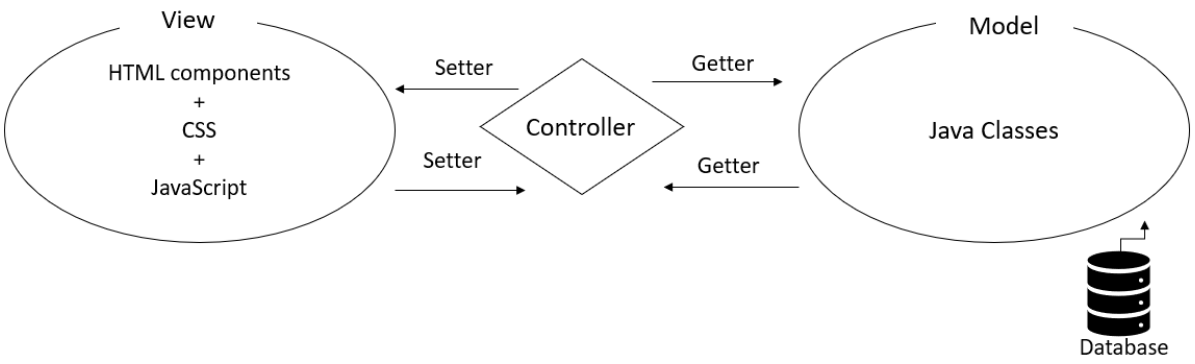

Figure 1.11: JavaServer Face web development framework usage of Model-View-Controller methodology

JavaServer Face web development framework usage of Java convention of *.setter()* and *.getter()* functions to control the interactions between frontend and backend component under the Model-View-Controller (MVC) methodology.

Lastly, the Django web development framework uses the same MVC methodology within the context of a Python programming language backend and the CakePHP of routing to enable interaction(Django Developers, 2022). Due to the Django web development framework being the framework that will be used for the various prospects of this thesis, a more in-depth discussion of its various characteristics will be discussed.

1.1.1 Django

Django is an entirely free web development framework with community support (Django Developers, 2022). This offers web developers that use the Django framework, a web development tool that is being developed based on the user experience due to its community support feature (Django Developers, 2022). Django being a web development framework, provides developers with an environment they can use for their full-stack development without cost (Django Developers, 2022; Stol & Ali Babar, 2010). Furthermore, free software like the Django web development framework with community support tends to have an edge in having more secure components and developmental cycles based on the quality of its version releases (Stol & Ali Babar, 2010).

If new requirements that include security features and usability needs of this framework tool that are contained within a comprehensive version release are informed by users (community members) (Stol & Ali Babar, 2010). These users tend to be diverse, from new users of the framework to experts who develop intensive web applications using the framework (Stol & Ali Babar, 2010). This tends to translate into a more comprehensive version release due to the feedback from the community of users regarding the needed functionalities a new version of the framework should have (Stol & Ali Babar, 2010). The free nature of Django does also contribute to enticing new users to use the framework (Stol & Ali Babar, 2010). However, is it worth noting that not all free software, regardless of purpose, translates into positive outcomes (Stol & Ali Babar, 2010).

A web development framework like Django can be very attractive for developers for various web applications, including bioinformatics web applications. Besides the points mentioned above, Django uses Python programming language to implement its backend logic (Django Developers, 2022). This is an important attribute of Django to make when discussing the web development framework's usefulness in developing bioinformatics web applications. Python programming language provides compelling libraries that enable the production of Python scripts that carry out bioinformatics needs (Sebastian, 2018). Python modules such as Biopython contain a series of libraries that a specifically geared toward addressing bioinformatics problems (Sebastian, 2018; Cock et al., 2009).

Given that developing bioinformatics functionalities in Python may require using files format in the biological field such as fasta, swiss and clustal file formats to name to carry out that particular functionality (Sebastian, 2018; Cock et al., 2009). For such functionality, the Python Biopython modules contain libraries like *SeqIO* and *AlignIO*, which can be used in developing functionality that involves the previously mentioned file formats (Sebastian, 2018; Cock et al., 2009). Furthermore, the Biopython module also contains libraries like *Seq* that contains functions such as *complement* (Sebastian, 2018; Cock et al., 2009). The *Seq* library and its various functions can take part in polynucleotide chains analysis, whereby the polynucleotide chain is represented in string format (Sebastian, 2018; Cock et al., 2009). The *complement* function will return the complement of that polynucleotide chain (figure 1.12).

```
install Python

pip install biopython --> Installation of the biopython module

#-------------------------------------------------------------

Python examples script for Seq library:

#-------------------------------------------------------------

from Bio.Seq import Seq

polynucleotide_chain = "ACATTCATGTGACCA"

input_chain = Seq(polynucleotide_chain)

complement_chain = input_chain.complement()

print(complement_chain)

#-------------------------------------------------------------

output >>> "TGTAAGTACACTGGT"
```

Figure 1.12: Implementation of the complement function of the Seq library from the Python Biopython module

The installation of the Biopython module using the Python *pip* package manager. Followed by importing the module library called *Seq* which takes in a polynucleotide chain in string format. The of complement this polynucleotide chain is created using *complement* function.

Figure 1.12, shows that to use the biopython module to used, the Python package manager *pip* needs to be used to install the module for local use. This is due to the Biopython library not being part of the already available and installed modules after Python has been installed, thus the need for the installation (Sebastian, 2018; Cock et al., 2009). Once installed various libraries from the Biopython module are then be imported where the functions available for that function can be called and used (figure 1.12). Furthermore, platforms such as Anaconda also provide more entising reason to use Python in the development of scientific studies (Anaconda Developers, 2022).

Anaconda is a free software package that is geared to facilitate the requirements of carrying out data science functionality (Anaconda Developers, 2022). Furthermore, the Anaconda package is available for usage in the Python programming language (Anaconda Developers, 2022). Functionalities such as data visualisation, predictive analysis, machine learning, neural networks and machine learning are offered by Anaconda (Anaconda Developers, 2022). For such offers, Anaconda can be considered a platform that contains all the previously mentioned resources on a single platform (Anaconda Developers, 2022). Such a platform as Anaconda being available in the Python programming language entices biological sciences developers to use Python as a programming language to carry out their desired scientific endeavours by using the Anaconda package (Anaconda Developers, 2022).

The previously mentioned packages that are available in the Python programming language make the Django web development framework which uses Python to carry out its backend functionality, even more enticing to biological developers (Django Developers, 2022). Therefore, this entails that Django's web application can be geared toward carrying out scientific functionality. As previously mentioned, the Django web development framework uses an MVC methodology where the controller in this application carries out a URL routing functionality (Django Developers, 2022). The routing function is used to connect the frontend events to their respective backend Python logic (figure 1.13).

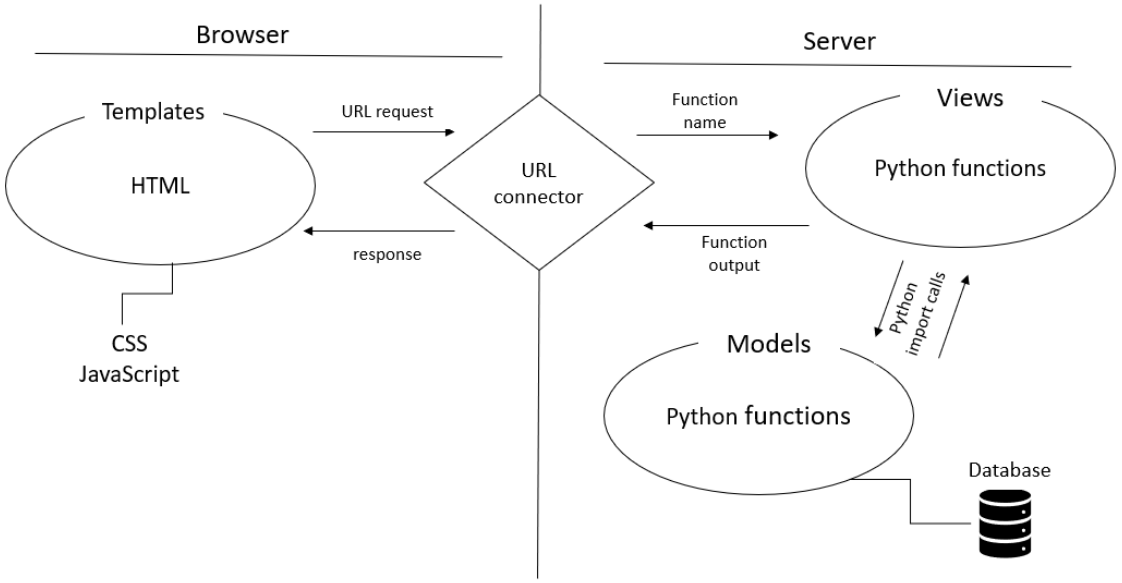

Figure 1.13: Django web development framework implementation of the Model-View-Controller methodology

The Django web development framework uses the MVC methodology, whereby the controller is actually a Uniform Resource Locators (URL) routing system. This URL routes requests from the frontend to the backend functions that handle the logic of that request. There the output of this function is sent back as the response.

In the Django web development framework, the view is composed of HTML, CSS and JavaScript scripts that will handle the frontend functionality of the web application in question (Django Developers, 2022). Typically in Django, the convention is that the HTML scripts are stored in a directory called *templates* (Django Developers, 2022). Therefore, the HTML scripts will form the template of the web application, which can then contain various embedded CSS and JavaScript components (Django Developers, 2022). The convention is also to store these CSS and JavaScript files in the directory named *static*, due to the CSS and JavaScript files being static during the run time of the web application (Django Developers, 2022). Furthermore, the HTML files will contain Django built-in tags and filters, which will give those HTML files a dynamic nature (Django Developers, 2022). This means that the various built-in Django tags and filters used in HTML files give Django the ability to be interactive and dynamic without the usage of JavaScript (figure 1.14).

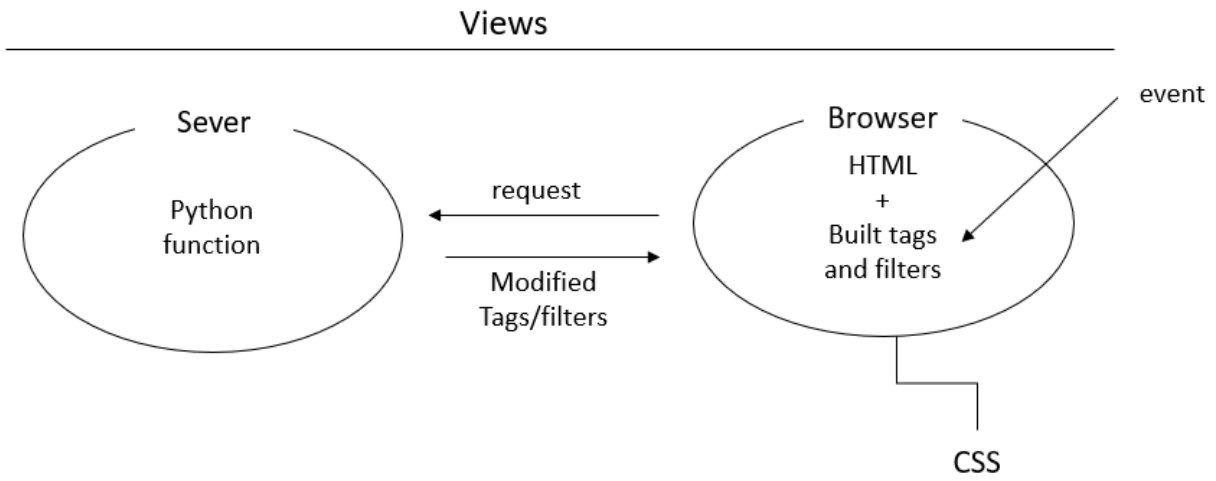

Figure 1.14: Interactions between HTML templates and Views Python functions in the Django framework

When an event from a web client is triggered in an HTML template file containing Django built-in tags and filters, the backend views the Python function relating to the given HTML template file will return a modified version of those tags or filters. Therefore this will change the HTML template file during run time.

Figure 1.14, shows how the Views being classified as handling frontend functionality has some of its components in the backend. Secondly, the Views component on the server-side uses Python logic to determine whether the called function will respond with modified tags or filters based on the type of event triggered by the web client (figure 1.14). However, the usage of built-in (embedded) Django tags and filters does not mean that JavaScript cannot be used to create interactive and dynamic HTML template files in the Django web development framework (Django Developers, 2022). Various JavaScript libraries can be used to explicitly design the interactive and dynamic structure of the Django web development framework (figure 1.13). The libraries in this instance are the JavaScript Jquery library and the JavaScript Knockout library, which can be used to achieve the same functionality as Django built-in tags and filters (OpenJS Foundation and jQuery contributors, 2022; JS Knockout Developers, 2022).

Due to both these libraries being built using JavaScript, they can be retrieved and placed in the *static* directory, which will enable the Django built web applications to access these libraries (Django Developers, 2022). The JQuery library carries its functionality by the request and response interaction between the web client, and the various components of the web application (OpenJS Foundation and jQuery contributors, 2022). JQuery library uses functions like the *.ajaxSend()*, *.ajaxError()* and *.ajaxSuccess()* to determine what action should be

taken based on the request and response (OpenJS Foundation and jQuery contributors, 2022). The *.ajaxSend()* function can be used to send requests to the Django backend Views Python functions at particular points of a given JavaScript script that relates to the event that a web client has triggered on the HTML template file of the web application (OpenJS Foundation and jQuery contributors, 2022; Django Developers, 2022). The *.ajaxError()* and *.ajaxSuccess()* functions can then inform the developer whether the request made by the web client has been a success or an error has occurred (OpenJS Foundation and jQuery contributors, 2022). The developer can then develop appropriate based on the functions previously mentioned to ensure that in case the web client is informed adequately about the request they have made (OpenJS Foundation and jQuery contributors, 2022).

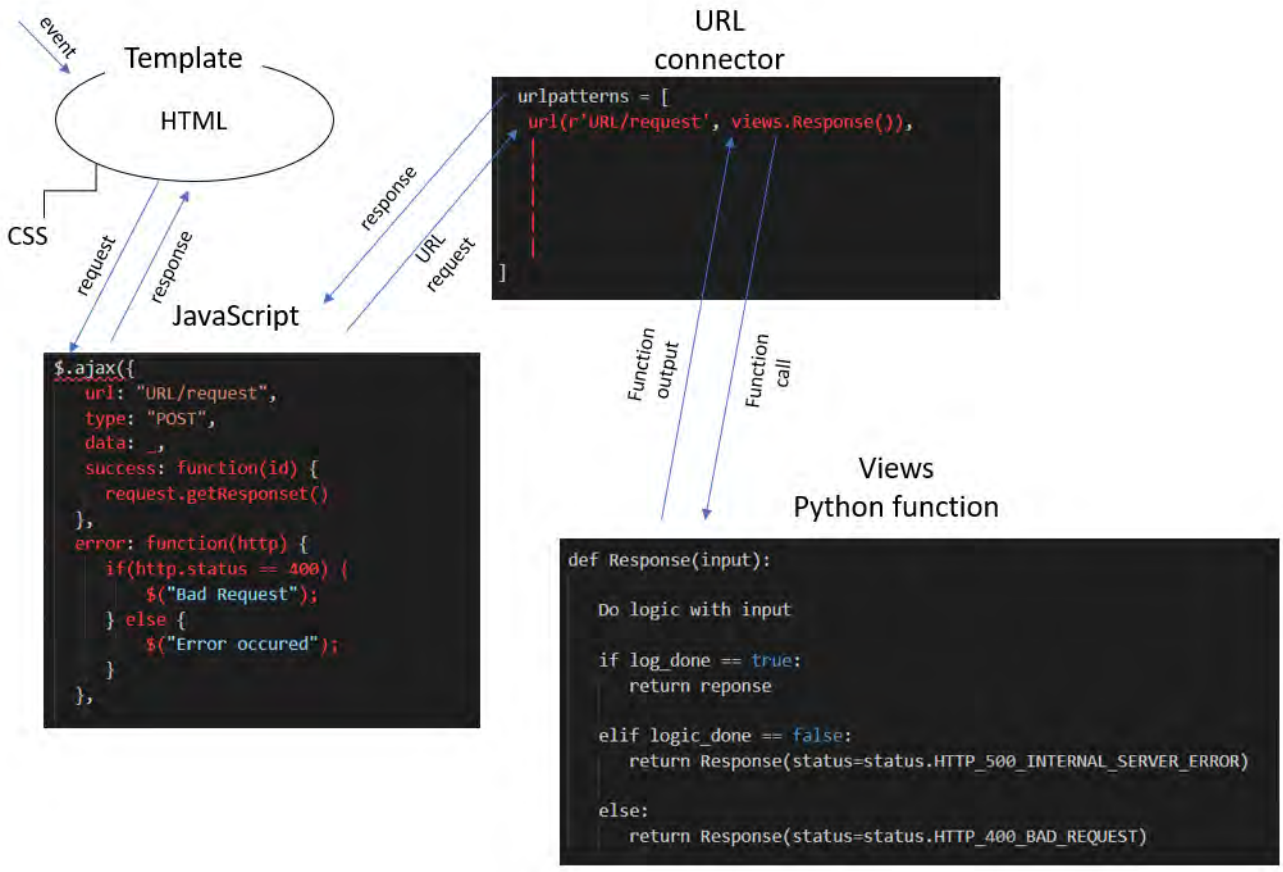

Figure 1.15: Example of how the URL connector enables interaction between HTML templates and Views in Django using JavaScript JQuery library

The JavaScript JQuery library with its ajax functions is used to exploit the Django URL connector functionality to create interaction between web client events triggered on the HTML template and the Views Python functions which will generate a response for the event.

Figure 1.15 shows how the *.ajaxSend()* function exploits the Django URL connector to access web application functionality from the Views Python functions while the *.ajaxError()* and *.ajaxSuccess()* functions categorise the output from the Views Python functions. The usage of JQuery shows how the construction of the Django web development framework does not constrain a web developer to solely use the tools offered by the framework (figure 1.15). This is further shown by the ability to use the JavaScript Knockout library previous mentioned to create interactive and dynamic HTML templates in the Django framework (JS Knockout Developers, 2022).

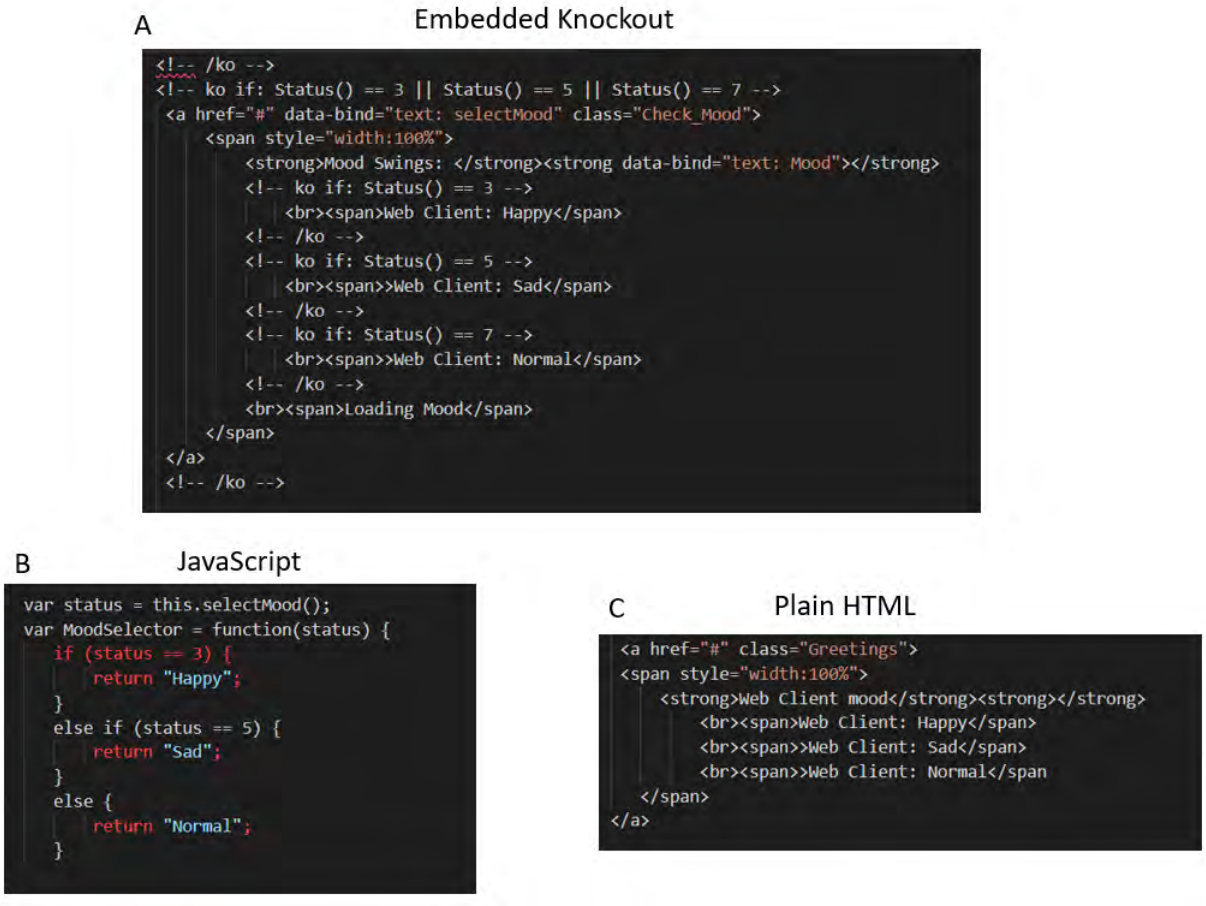

Figure 1.16: Example of embedded Knockout tags within HTML file along with the related JavaScript code

An example of how a dynamic HTML file can be created using the JavaScript Knockout library based on the status, which will display the associated Knockout section.

A: Shows how embedded Knockout tags are constructed within dynamic HTML files. The knockout data-bind attribute is added to observe the events associated with that HTML tag.

B: The JavaScript code that will be executed with the values received from the embedded Knockout code.

C: Static HTML code, for comparison purposes.

The Knockout library's ability to add its attributes both to HTML tags and also into sections of the HTML file enables the Knockout library developers to create dynamic parts of an HTML file (JS Knockout Developers, 2022). This feature of Knockout, gives web applications that use the library to form parts of a given web page interactive or dynamic ability while the rest of the web page can remain static (JS Knockout Developers, 2022). Figure 1.16, shows how the JavaScript Knockout library is embedded within an HTML file and how the *data-bind* attribute is used as an event triggered observer. The Knockout library enables logic to be added to the Django HTML templates directly utilising the concept of binding (figure 1.16). Binding is a concept whereby the Knockout library functionality attributes can be bound to HTML tags, enabling the Knockout attribute also to be triggered when the HTML tag is triggered (JS Knockout Developers, 2022).

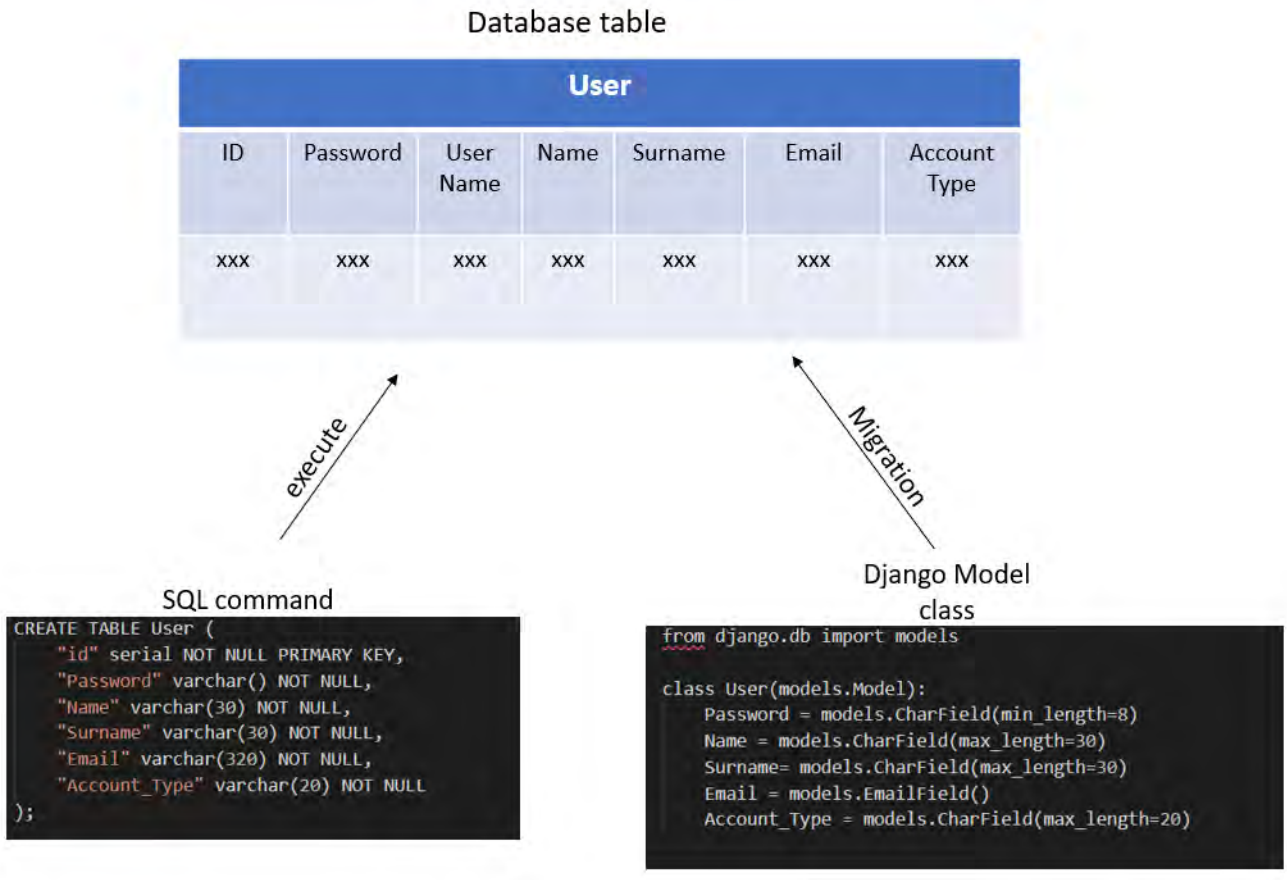

Figure 1.17: Example of Django Models class table creation relative to SQL command

The create of a table within a database both using the Django web development framework Model class and using the typical SQL table create command.

All the above-mentioned techniques can be used in the Django web development framework to generate interactive and dynamic web applications, while these techniques offer different implementation styles. The various tools that can be used to implement a web application shows the versatility of the Django web development framework.Figure 1.17, shows how the Django web development framework uses the Model class to create a table in a database relative to how a table would be created using SQL command to result in the same database table creation. Furthermore, the Django web development framework uses migration to move changes that have been made to the Models onto the database (figure 1.17). The way the Models classes enables Python developers to perform interactions with the database without the need to have an in-depth understanding of SQL (Django Developers, 2022). This is due to the understanding that the Django Models structure will provide sufficient information to perform these interactions with the database with minimal knowledge of SQL, which the Django Models is based on (Django Developers, 2022).

Lastly, the Django web development framework uses the Models section of its structure to set the attributes of the tables contained in the database (Django Developers, 2022). The Models will be Python classes that will represent the various tables in the database, with the content of these classes being the attributes of the columns of that given database table (Django Developers, 2022). Furthermore, these Python classes can be used to query the database by populating and retrieving the given table using the Django database functions *.objects*. The Django database *.objects* function has query calls such as *.get()*, *.filter()* and *.order_by()* to name a few (Django Developers, 2022). The *.get()* will query the table in question to determine whether the given parameter matches an entry in the database table (Django Developers, 2022). The *.filter()* will return database table enters which have been filtered based on the given parameters, while the *.order_by()* query will return enters in given parameters order (Django Developers, 2022).

Due to the way the Django web development framework deploys Models to generate database interaction with the web applications, ensure that malicious activity such as SQL injection can not be conventionally carried out (Django Developers, 2022). For all the functionality to be consolidated from the HTML template, Views, Models and the static files, a *settings* Python file is used whereby all the directory of these files and their related functionalities are initialised (Django Developers, 2022). This *settings* Python file will contain information about the database, all the associated files and their directories to ensure the web application has access to all these dependencies (figure 1.18).

```
from pathlib import Path

BASE_DIR = The directory which will form the root of the web application contents

ALLOWED_HOSTS = [A list of the different IP addresses
                 which have access to the web application]

INSTALLED_APPS = [
    all the applications that the web application uses
]

MIDDLEWARE = [
    string list of the middle ware used
]

ROOT_URLCONF = 'URL Connector information'

TEMPLATES = [
    {
        Python directory contain HTML template information
    },
]

DATABASES = {
    'default': {
        Database access information
    }
}

STATIC_URL = 'Directory that has the static files'
```

Figure 1.18: Example of information contained in the Django setting Python file

The contents of the *settings* Python file used by the Django web development framework. This file will initialise all the information that will be used throughout the web application.

Additionally, Django provides a user interface called Admin that can be used to manage the database content once the models have been successfully migrated (Django Developers, 2022). Furthermore, Django does support non-rational databases like Mongo-DB, giving Django Models back end versatility, although such functionality is offered by a third party (Django Developers", n.d.). Lastly, Django does provide back end security by offering to convert database fields like passwords into hashes, thus providing encryption (Django Developers, 2022).

1.2 Hosting

Once a web application has been developed, the web application needs to be hosted for it to be accessible to web clients (Ibrahim et al., 2021; Nguyen, 2017). All web development frameworks such as CakePhP, JFS and Django offer its developer the ability to host the web applications locally (del Pilar Salas-Zárate et al., 2015). The local hosting of the web application, however, does not mean that web clients can have access to the web application (del Pilar Salas-Zárate et al., 2015). The locally hosting of the web application only servers for testing of the web application (del Pilar Salas-Zárate et al., 2015).

To host a web application, a web server needs to be set up, and the type of web server software used will determine various aspects of how web clients will interact with the web application (Ibrahim et al., 2021; Nguyen, 2017). Firstly the type of web application will dictate what web server is most suitable to ensure, as previously mentioned, the most optimal web client experience (Ibrahim et al., 2021; Nguyen, 2017). This is due to a web application, for instance, that does a lot of computation relative to a web application that is for data extraction will have different web server and hardware requirements to ensure performance is maximised (Ibrahim et al., 2021; Nguyen, 2017). Enabling access to a given web server is a software issue, the performance requirements of the web application thus become a hardware issue (Ibrahim et al., 2021; Nguyen, 2017). There are various web server software and supporting hardware for different web application requirements (Ibrahim et al., 2021; Nguyen, 2017). For instance, the hosting of bioinformatics web applications can be done in various ways, depending on the function of the bioinformatics web application. Some bioinformatics web applications may need to be hosted on a web server that has hardware that enables intensive computation. In contrast, others may need storage capability or rendering capability (Ibrahim et al., 2021; Nguyen, 2017).

However, a single web application may need to be hosted on a web server with all the computation, storage, and rendering capabilities to carry out its functionality. With this in mind, a solution that can be offered to such web applications is a distributed web server system (Ibrahim et al., 2021; Nguyen, 2017). A distributed web server system can be described as a web server that has components that participate in the web application functionality that are not found in signal server (Ibrahim et al., 2021). This solves the issue of having a web application that has different functional requirements by implementing the web application on distributed web servers (figure 1.19).

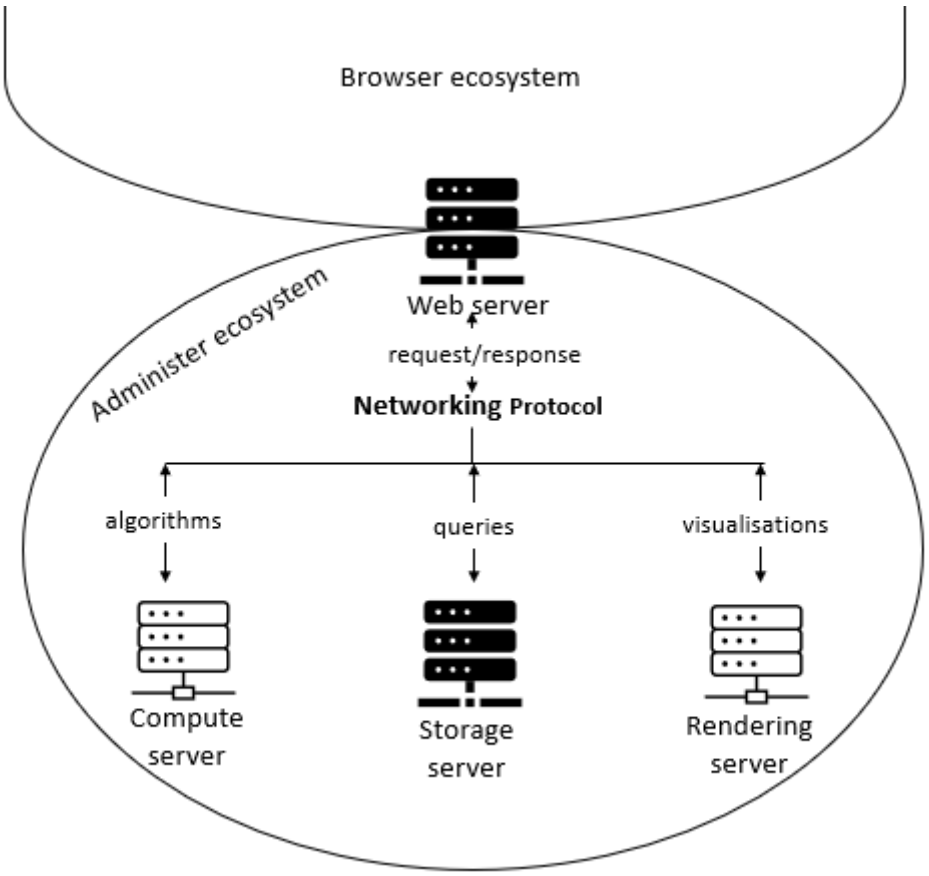

Figure 1.19: Demonstration of a web application that hosted on the a distributed system

A web application that is hosted on a distributed system where by the requests with specific requirements are sent to the server which has the capability to service those requests. Whereby the compute server will handle the requests that require computation, storage server will handle request that require storage and the requests that need to be rendered will be sent to the rendering server.

Figure 1.19 shows how the principle of a distributed web server can to applied to a single web application. In this instance the requests sent by the web clients relating to a particular functionality can be sent to the server which handles that functionality (figure 1.19). A distributed server system can also be called a cluster system with each server in the system being a node that handles the functionality that is designated to it that node (Ibrahim et al., 2021). The usage of cluster system in implementing can be due to improving response time to web clients requests (Ibrahim et al., 2021; Nguyen, 2017). Additionally, this can be an understanding by the web application developers that the requests that the web application will receive will require particular resources. Thus the implementation of the distributed fashion to handle those types of requests which require the additional resources (Ibrahim et al., 2021; Nguyen, 2017).

Given the performance potential in distributed web servers, some considerations need to be made which can either increase or decrease this potential (Ibrahim et al., 2021; Nguyen, 2017). Some of these considerations are network costs, which is how much time is consumed by transporting requests between the different distributed web servers before a response is sent to the web client (Ibrahim et al., 2021; Nguyen, 2017). This issue can be mediated by the implementation of a caching system or by introducing redundancy within the distributed system (Ibrahim et al., 2021; Nguyen, 2017). However, this approach is limited to data orientated web applications. Therefore, computation oriented web applications that use a distributed web server will need different optimisation approaches (Ibrahim et al., 2021; Nguyen, 2017).

1.2.1 Web server software

There is various software available to ensure a web server is accessible on the web, namely NodeJS, Nginx and Apache Web Server, to name a few (Ibrahim et al., 2021; Nguyen, 2017). NodeJS web server is JavaScript built software that is meant to form web servers that host web applications that use JavaScript as their backend (Ibrahim et al., 2021; Nguyen, 2017). The configuration of the NodeJS web server is done by importing the HTTP (Hypertext Transfer Protocol) core module, which is used in the creation of the actual NodeJS server (figure 1.20).

```
var http = require('http');

var server = http.createServer(function (req, res) {
    if (req.url == URL) {

        // execute logic
        res.write(logic output);
        res.end();
    }
```

Figure 1.20: Setting up a NodeJS web server

Importing the HTTP (Hypertext Transfer Protocol) module is then used to create the webserver. The web server function will have the request and response parameter, whereby if the web client sends a request on a particular web page (URL), the appropriate logic and response will be sent.

Figure 1.20, shows how the HTTP module is used to create the NodeJS web server. The first thing to notice is that this webserver is written in JavaScript (figure 1.20). The HTTP module server creation function used will have the request and response parameters (figure 1.20). The request parameter has a function called *.url*, this function will return the URL of the web page or URL of the event triggered to the webserver (figure 1.20). The return of the URL is then matched with all the URL requests which are expected to be made, and the logic of each of those requests can be executed and the appropriate response written back to the web application (figure 1.20).

NodeJS can be scalable by using its single thread functionality by running the threads concurrently, giving a higher performance by using the threads to handle the different requests to the webserver from web clients (Ibrahim et al., 2021; Nguyen, 2017). On the hand, NginX uses a concept of thread pools, whereby a master and workers hierarchy is set as it is an event-driven web server that distributes the load (requests) (Ibrahim et al., 2021; Nguyen, 2017).

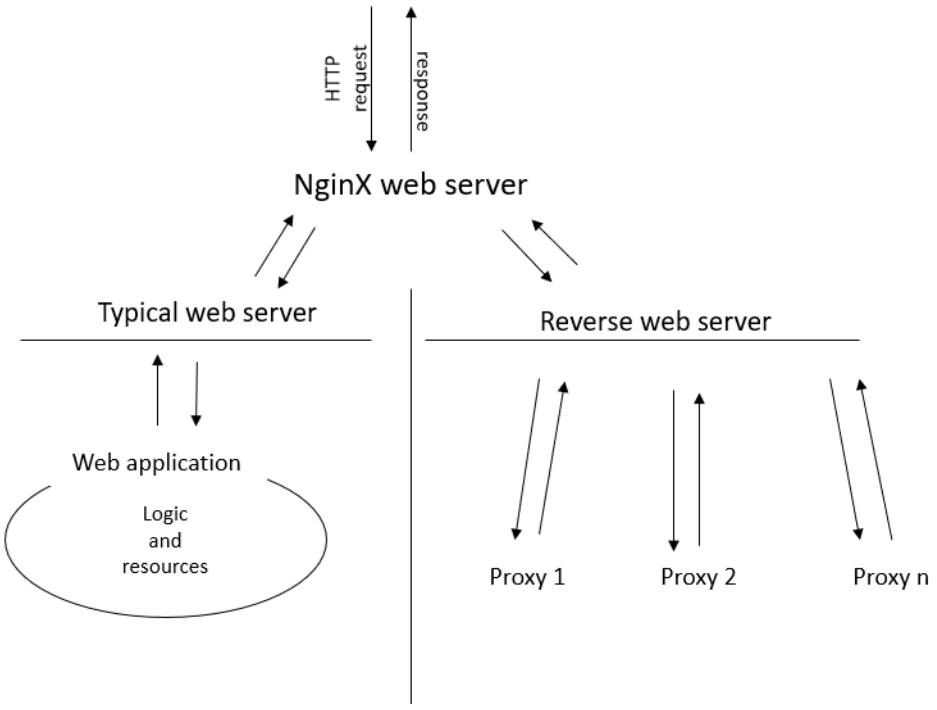

Figure 1.21: NginX web server as a standard web server and as a reverse web server

The dual functionality which a NginX web server offers by being a typical web server which publishes a web application on the web, handling its request and response while also being a reverse web server directing incoming web requests to the relevant web server.

NginX web server software is popular due to its diverse application, and suitability (Ibrahim et al., 2021; Nguyen, 2017). NginX web server can be used as a typical web server hosting software that enables a web application to receive HTTP requests and also respond to those request (Ibrahim et al., 2021; Nguyen, 2017). However, an Nginx web server can also be used as a routing or reverse web server that can receive HTTP requests and send those requests to the relevant servers (figure 1.21).

Figure 1.21, shows the different functionality a NginX web server can have. An NginX web server can be used for web applications that have a high volume of requests due to its architecture's ability to service multiple connections (Ibrahim et al., 2021; Nguyen, 2017). Due to all the features of the NginX web server, it can be used in cluster systems since it can use its reverse proxy server functionality to distribute the request loads appropriately (Ibrahim et al., 2021; Nguyen, 2017). Furthermore, the cache system deployed by NginX web server improves performance by ensuring that requests that have responses have been cached (Ibrahim et al., 2021; Nguyen, 2017). The cache of responses is then made available to the web client without the web application executing the logic required to generate a response for that request (Ibrahim et al., 2021; Nguyen, 2017). The root of the reverse proxy functionality can be perceived as the master, which distributes resources from the thread pool of workers to ensure load balance in terms of requests being handled from the web clients (Ibrahim et al., 2021; Nguyen, 2017). The NginX web server is typically installed on machines that use a Linux operating system.

Lastly, Apache functions on the same conceptual setup as NginX, whereby it provides web clients access to a given web application, making both Apache and NginX the most widely used web server software on the internet (Ibrahim et al., 2021; Nguyen, 2017). Furthermore, Apache is also typically deployed on machines that use the Linux Operating system. However, Apache uses multi-processing modules instead of workers to handle the various requests that are sent to the webserver (Ibrahim et al., 2021; Nguyen, 2017). Therefore, the Apache webserver is process-driven instead of being event-driven like a NginX webserver (Ibrahim et al., 2021; Nguyen, 2017). The multi-processing module feature of Apache means that every request results in a new process being formed, making an Apache web server more flexible (Ibrahim et al., 2021; Nguyen, 2017). However, this flexibility of using processes does affect web application performance during high traffic on those web servers as more resources will be required to deal with the processing being formed (Ibrahim et al., 2021; Nguyen, 2017). As a result the selection of the most appropriate webserver to deploy between Apache and NginX will depend on the level of traffic that the given web server will experience (Ibrahim et al., 2021; Nguyen, 2017).

NginX being an event-driven webserver, will maintain its performance during high traffic while an Apache will decrease in performance (Ibrahim et al., 2021; Nguyen, 2017).

1.2.2 Web server hardware

A factor that will not be addressed in the paper but is also an essential consideration in hosting web servers, particularly in distributed systems, is the type of hardware that the webserver software will be operating on. Briefly, the kind of hardware tends to typical belong to two general categories, which are compute servers and storage servers. The specification of those categories can vary for more specific applications (Shetty & Ganashree, 2020). Generally, compute servers in a distributed system in regards to web application tends to be the server handling the requests that require significant computational throughput (Shetty & Ganashree, 2020). Therefore compute servers can be characterised by having high Central Process Units (CPUs) with multiple cores, various cache levels and Random Access Memory (RAM) to enable the capacity for this high computational throughput (Shetty & Ganashree, 2020). Figure 1.22, shows how a multi-core CPU tends to be constructed to enable high computational throughput.

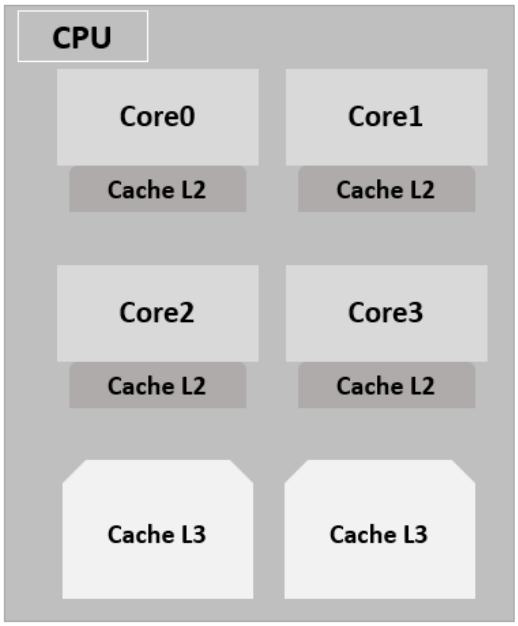

Figure 1.22: Demonstration of a AMD Phenom multi-core Central Processing Unit

The AMD Phenom Central Processing Unit (CPU) has multiple cores where each core can do computation. The L2 and L3 cache systems are used for workload storage support to the cores. The L2 cache is faster compared to the large L3 cache. The L2 cache ensures that computation workload is readily provided to the cores, ensuring the cores are always computing.

The usage of a cache system is vital for the computational throughput since a multiple-core CPU is only useful if the cores in that CPU are consistently performing tasks (Shetty & Ganashree, 2020). Therefore the implementation of a cache system enables tasks to be transferred to the cores rapidly (Shetty & Ganashree, 2020). Completion of the tasks by the CPU cores will require the other tasks to be sent back to the processes that sent those tasks to the CPU (Shetty & Ganashree, 2020). The L2 cache is a smaller yet faster cache type relative to the L3 cache (Shetty & Ganashree, 2020). There is an L2 cache for each core, while the large L3 cache, in this case, is used to store awaiting tasks once memory to transfer those tasks to the L2 cache is available (figure 1.22). RAM also functions as a cache system for all the processing being executed on the system, ensuring that processes have access to memory to store their various executions requirements (Shetty & Ganashree, 2020). If the RAM is not present, the processing being executed on a given system will need to use the hardware storage which is typically slower and can be congested with other processors also trying to access this memory location (Shetty & Ganashree, 2020).

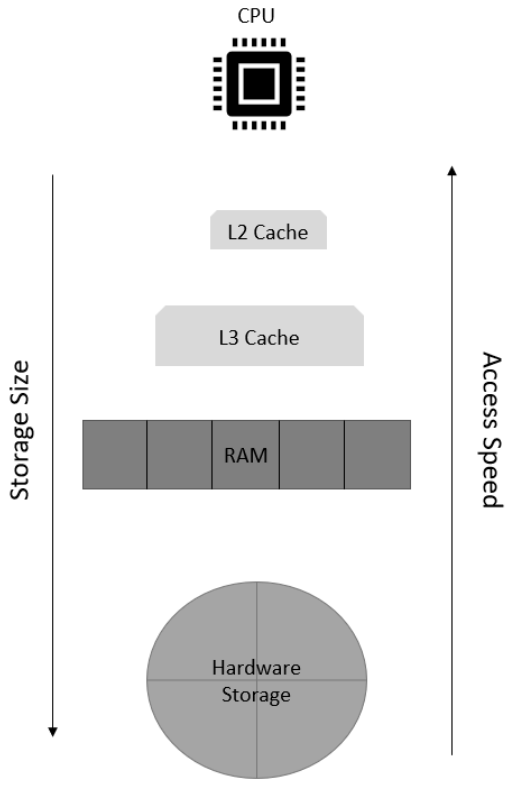

Figure 1.23: Storage types relative to access speed

Different storage types are typically implemented on a computing system. These include the hardware storage that typically has the largest storage capacity, followed by the Random Access Memory (RAM), then the L3 caches, and then the L2 caches. However, in this instance, the larger the memory, the slower the access speed.

For an optimised system whose function is for high computational throughput, hardware storage supported by a vast RAM and cache systems is required (figure 1.23). This ensures that a bottleneck is not created to memory requirements of executing tasks (figure 1.23). This is due to the relation between access speed, meaning how quickly read and write can proceed on a given memory space and storage size, which is the amount of data that can be stored on a given memory space (figure 1.23). Storage servers, on the other hand, tend to house any data from audio, text, image and any other type of format, typically in large quantities (Shetty & Ganashree, 2020).

It is worth mentioning that the are different hardware attributes that can be used for storage servers depending on their functions (Shetty & Ganashree, 2020). In regards to web applications, storage servers tend to be large storage databases that will be used by the web clients of that particular web application implemented in a distributed system (Shetty & Ganashree, 2020). Therefore the hardware attributes of such a storage server will need to be hardware that is tolerant of intensive read and write executions since web clients will be implicitly doing such executions when interacting with the database (Shetty & Ganashree, 2020). It can be the case where a web application serves as a location for web clients to store data (Shetty & Ganashree, 2020). In such instances, hardware that is tolerant of write-intensive execution is important (Shetty & Ganashree, 2020).

Lastly, an important component of the hardware used in a storage server is the data recoverability capacity of the hardware (Shetty & Ganashree, 2020). Many factors inform this, but it is important to ensure that the hardware used and its facilitating software enable data recoverability (Shetty & Ganashree, 2020). The possibility of the storage server failing can be mediated by having back-ups from other storage servers (Shetty & Ganashree, 2020). However, ensuring that the selected hardware and its facilitating software used for storage is geared towards the purpose of the server and has the characteristics that make data recovery easier (Shetty & Ganashree, 2020).

Chapter 2

Traceability of web applications, tools and data

Various techniques are available for archiving traceability of a given developmental project. As mentioned in the introduction, one of the ways to perform traceability is through documentation. Documentation does not only provide value in software development but can also provide value in any setting that is in the process of change where various changes are introduced into the process (Tian et al., 2021). For research like Research Unit in Bioinformatics (RUBi) that use distributed computer system (cluster), whereby each part of the system has a different function, it gives value to document such a system for reproducibility and expansion (Tian et al., 2021). Reproducibility ensures that in case of any failure in the system, the documentation can be used to understand the implications of the failure. Furthermore, to determine what type of intervention is necessary to resolve the issue (Tian et al., 2021). In regards to expansion, documentation can show what type of resources, whether hardware or software, are needed to improve the system in ways that give the best collective performance of the system (Tian et al., 2021).

Furthermore, traceability can also be used in other instances, such as the development of software or algorithm (Tian et al., 2021). Due to typical software or algorithm development having different types of implementation prospects and being prone to change (Tian et al., 2021). The reasons mentioned previously show how traceability can be used to keep track of such developmental process (Tian et al., 2021). For such developments, written documentation can be tedious as the writer of such document has to be mindful of various aspects (Zolkifli, Ngah, & Deraman, 2018; Kubilius, 2014). On that bases, version control software provides greater insight and functionality in such development project to serve as the documentation process (Zolkifli et al., 2018; Kubilius, 2014).

2.1 Documentation

Documentation is typically in written format but can also be in other formats such as audio and videos formats(Tian et al., 2021). In the context of this section, documentation was done in a written format. The documentation was done to trace the various components of the RUBi cluster system, whereby each node in the system serves different functions. Documenting the nodes in the RUBi cluster system will give information regarding functions carried out on those nodes and the various tools those nodes use to carry out their set functions.

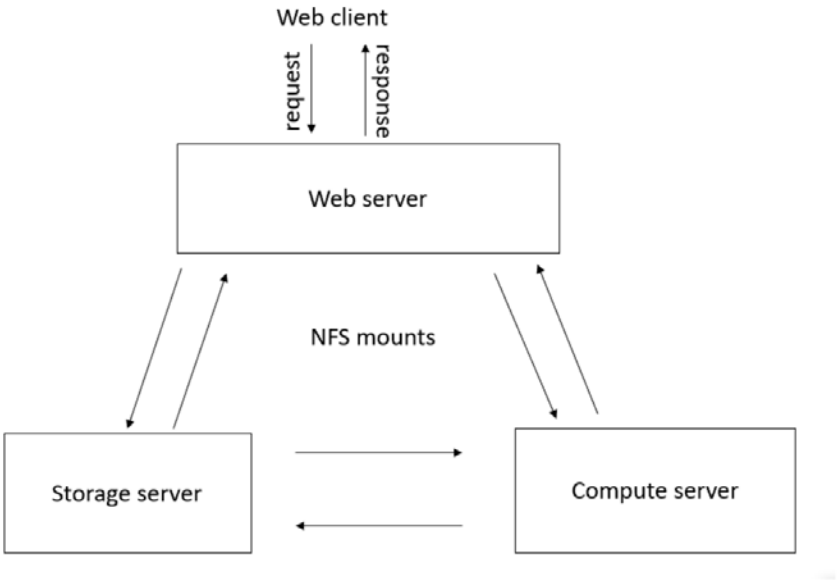

Figure 2.1: Distributed web server system

The figure displays where the data used by the web server will be stored and where the request that required computation will be executed. The different components of the distributed system communication via a Network File System (NFS), transporting the data via the network.

Figure 2.1 gives a simple view of how a typical web server that is used in a distributed system, like RUBi web servers, are situated. The various web applications hosted by RUBi carry out their various functions on the distributed system (figure 2.1) All the nodes in the RUBi cluster systems communicate with each other via Network File System (NFS). The usage of NFS means that the nodes in the cluster system need to be connect to some type of network system (Zakarya, Rahman, & Ullah, 2012). The network system will give each node a unique IP address that can then be used by the nodes within the to system to communicate with each other via the NFS client and server mounting system (Zakarya et al., 2012). The NFS client and server mounting systems enable to the different nodes in the RUBi to directly communicate with each other (figure 2.1).

As previously stated, the documentation of the storage server will give RUBi administrators the ability to reproduce all the functional structures found on the storage server. The documentation was stored on a RUBi storage server called *Jabba* and was written in a text file named README, as that is the conventional name used for files that contain information. This storage server contains software tools used by web applications, compute servers, and various protein data used by RUBi web applications. The documentation was done in text format and can be accessed by all the NFS attached servers (figure 2.1).

2.1.1 Methodology

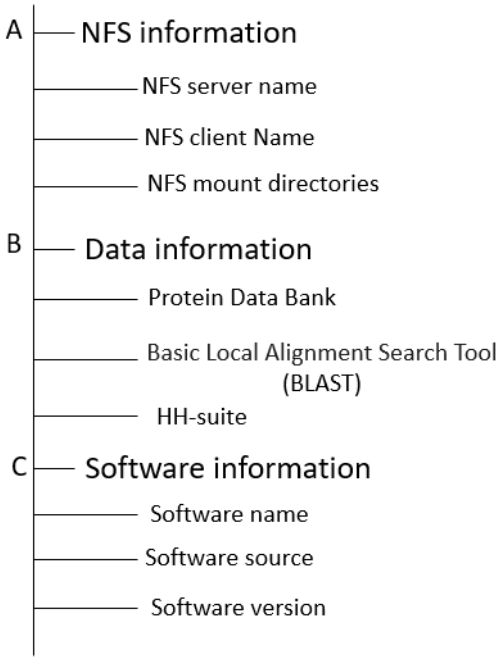

Figure 2.2: Documentation structure

The documentation structure of the the RUBi *Jabba* storage server:

A: The Network File System (NFS) will show all the NFS information used throughout the RUBi cluster system in relation to the *Jabba* storage server.

B: The information regarding the data that is stored on the *Jabba* storage server.

C: The software tools that are used by the RUBi cluster system and the information relating to the those software tools.

To perform the documentation process of the RUBi cluster system, an investigation into all the nodes used and the contents of those nodes had to be undertaken. This methodology will then provide information on a node level to determine node functionality and contents based on its allocated cluster system function. This is particularly important in the context of RUBi, as the development of the cluster system was not documented. Therefore no assumptions can be made about the nodes in the system as there is a possibility that the nodes can have different content and functions given the age of the RUBi cluster system.

The documentation was done in a particular structure to ensure the relevant information was grouped appropriately (figure 2.2). The documentation firstly will include the NFS mounts of the storage server (figure 2.2). This will ensure that all the information regarding the mounts required is available when a storage server migration occurs. Providing the new storage server with all NFS mounts to the revelation locations. Additionally, the administrator can be aware of other servers the storage server interacts with within the network. *Jabba* being a storage server, various data used by RUBi web applications are also stored on that server (figure 2.2).

The data stored on *Jabba*, is protein data that is composed of structural, DNA and RNA data of proteins. The protein data such as Protein Data Bank (PDB) structures can be retrieved using tools like Blastp, whereby the whole PDB database can be retrieved and stored locally. The documentation will include the tools used to retrieve the protein data due to the continuous release of new protein data. It is worth noting that this storage server contains data that can be retrieved from third party sources. Lastly, the *Jabba* storage server also houses, as previously mentioned, the software tools in the web application.

Due to the storage of software tools on that particular server, it is important to document the names of the software tools, the versions of those software tools and the sources which those tools were retrieved from. However, this could not be done for all software tools, as RUBi has legacy software tools that were not initially documented. As a result, some of the versions and sources of these tools could not be traced. For tools that could be traced, it is important in terms of updating and retrieving those tools that the documentation brings value in ensuring that the same sources are used for consistency and traceability within the system.

2.1.2 Results

The value in this is knowing which servers may be affected by any fault on the storage server. Secondly, if the storage server requires any maintenance, the administrator can use the given NFS mounts directives provided in the documentation to reproduce the storage server after maintenance. In regards to information about the data stored on the server, it provides the administrator with information about that data on *Jabba* and, with knowledge of the RUBi cluster system, can then determine which web application uses such data. Additionally, this can inform of the places the administrator should look if a data issue were to raise from the web application.

The advantage of documenting the data and the data retrieval process is that if a complete failure occurs on the storage server, the data can be retrieved from those third party sources. However, the reliance on third party sources can itself be problematic if those third party sources are unavailable (Bauer, Heinemann, & Deissenboeck, 2012). Therefore, as previously mentioned, it is important to ensure that other tools have been identified that can carry a similar function in an instance where the third party sources fail.

Same can be stated in regards to the documenting of the software tools. It is advised that the software packages are back up to ensure that if the sources are not found, the packages can be used to "re-build" the tools as the authors of the software tools can decommission the source. It is also worth noting that the packages can also be corrupted. Therefore, replacement tools should be considered for an unfortunate instance where a fault occurs, corrupting the packages resulting in them not being retrieved.

Lastly, the documentation process of this storage server highlighted how advantageous having in house tools deployed in the distributed system is due to the control and reliability of using in house tools (Bauer et al., 2012). However, the development of such tools requires resources that units like RUBi may not have. Therefore, third party tools enable relatively high productivity in an environment that has limited resources, but the organisation must be conscious of the risks in such a reliance (Bauer et al., 2012).

2.2 Version Control Systems

Version Control Systems (VCS), which will be referenced as repositories in recent history, has been a vital resource in the software development life cycle (Zolkifli et al., 2018; Kubilius, 2014). Repositories provide various offerings for software developers, and these can be from collaborating in software development, software accessibility and, in the context of this chapter, the version control of software and archiving (documenting) of software (Zolkifli et al., 2018; Kubilius, 2014). Typically, repositories use a few conventions which determine the type of functionality that will be executed on the repository (Zolkifli et al., 2018; Kubilius, 2014). The convention that is typically used is the *Git* framework, which is a software tool that can be used to facilitate interactions between the local code being worked on and the repository (Zolkifli et al., 2018; Kubilius, 2014).

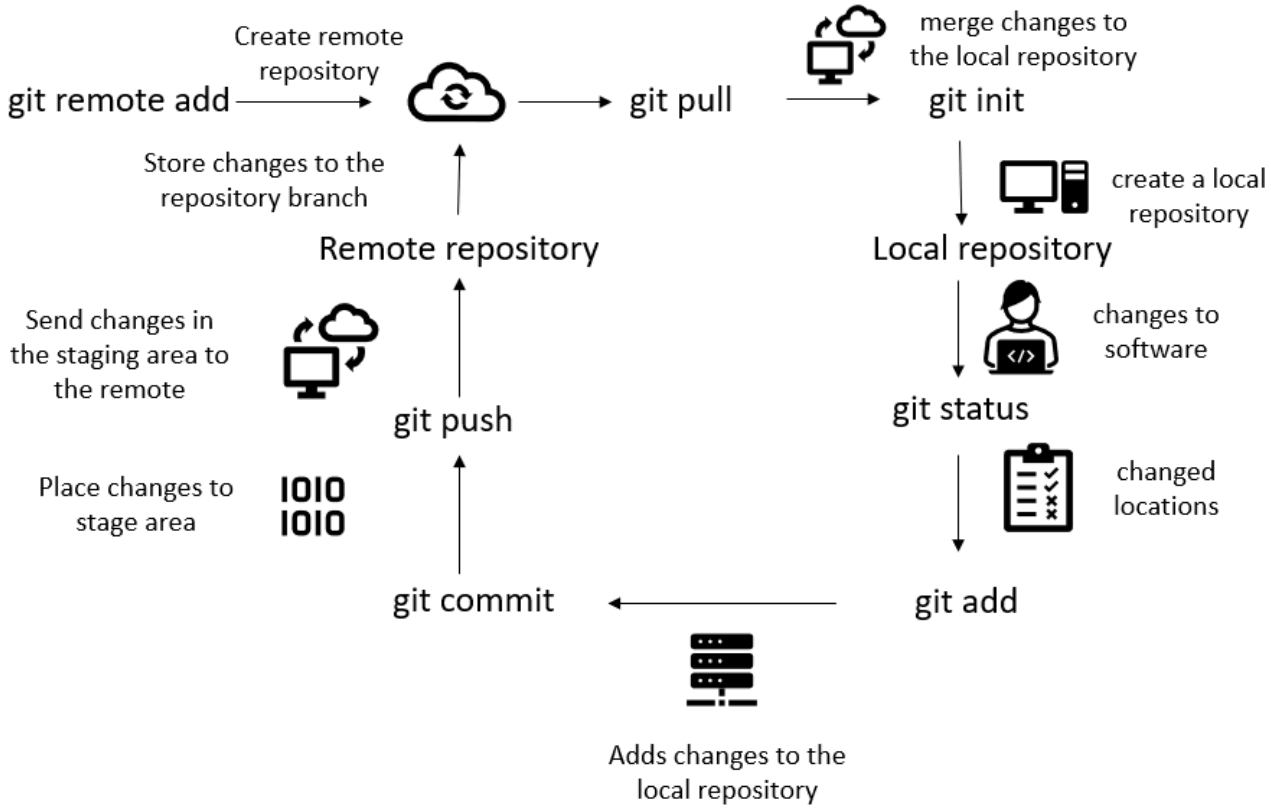

Figure 2.3: Setup and interaction with a repository using Git commands

The use of *GIT* commands to set up a local repository using the *git init* which is used to develop software. Whereby the process of the development is added to the working tree using *git add* and the changes can be places in the staging area using *git commit*. Lastly, the changes are pushed to the remote using *git push*, the remote is created using *git remote* and the contents on the remotes placed in the working tree using *git pull*.

Figure 2.3 shows how a simple *Git* repository is set up, and all the *Git* commands used to interact with the repository which implements VCS. To use the *Git* to interact with a repository, there are a few requirements to enable the intended interaction with the repository. Firstly a local repository needs to be set up where the various changes can be staged using the *git init* command (figure 2.3). Secondly, a local repository will need to be connected to the repository using the *git remote* command, which will function as a remote that stores the changes (figure 2.3). Once a local repository which is connected to its remote has been set up, various Git commands can be executed to perform different functions.

There is some terminology used to describe the different functions and instances in a given VCS. To have a simple understanding of a VCS, one must view software development as a process of continuously adding changes to a given set of files (Zolkifli et al., 2018; Kubilius, 2014). The VCS then keeps track of these changes whereby the different changes can be described as different versions (Zolkifli et al., 2018; Kubilius, 2014). In the VCS world, these different versions are called *diffs* and each point of difference between the versions is called a *hunk* (Zolkifli et al., 2018; Kubilius, 2014). All these different changes are called *changeset*, to capture these changes within a VCS is called a *commit* whereby putting these as a version in the VCS is called *committing* (Zolkifli et al., 2018; Kubilius, 2014). *Git*, as previously mentioned, offers commands which enable the management of all these different parts of a VCS (Zolkifli et al., 2018; Kubilius, 2014).

Some of *Git* commands which a worth noting are *git diff*, *git status*, *git branch*, *git reset*, *git checkout*, *git merge* and *git clone*. The *git diff* command will show the staged changes, while *git status* command will show which *changesets* have not been staged (Zolkifli et al., 2018; Kubilius, 2014). On the other hand, the *git branch* command creates branches whereby, for example, different functionalities of software can be developed on those different branches within the VCS (Zolkifli et al., 2018; Kubilius, 2014). The *git reset* command moves the head pointer to a different branch, while the *git checkout* command will move the head pointer to a specified branch (Zolkifli et al., 2018; Kubilius, 2014). The *git merge* command enables the differences of given branches to be moved into a specified branch (Zolkifli et al., 2018; Kubilius, 2014). Lastly, the *git clone* command will retrieve a copy of the contents on a remote repository into a local repository. It is worth noting that the *git merge* command can result in conflicts whereby the *changesets* of different branches occur on the same location (Zolkifli et al., 2018; Kubilius, 2014). However, this highlights, amongst other valuable attributes of a VCS using *Git* the collaborative potential offered by VCS along with the mentioned archiving ability.

Due to the ability to push changes during the developmental cycle of a given software into a repository and revert to a particular version enables the ability to trace the source code development of that given software (Zolkifli et al., 2018; Kubilius, 2014). Furthermore, remote repositories tend to be offered by third-party vendors such as GitHub and Bit Bucket, which enable a remote to be created on their platforms, to mention a few (Zolkifli et al., 2018; Kubilius, 2014). This gives developers a remote repository functionality which can then be used to implement the above-mentioned processes (Zolkifli et al., 2018; Kubilius, 2014).

2.2.1 Methodology

RUBi has used the mentioned third party vendors where Git Hub (https://github.com/ RUBi-ZA) is for the publicly available repositories such as projects as MD-Task, Mode-Task and JMS software tools, to mention a few. As a platform that hosts private repositories, Bit Bucket is used as a repository for software tools that RUBi has not made public. These platforms were primarily used by RUBi for software archiving for some software and software publications (accessibility). However, the archiving feature of these platforms was not done for all the software developed by RUBi, making this task of archiving software essential since RUBi stores its software on its various servers. As previously mentioned, faults do occur on servers, and without using a platform like Git Hub or Bit Bucket, there is a risk that all software may not be recovered once the fault has been fixed (Kubilius, 2014). If such a fault were to occur, RUBi would have to rebuild all its web applications and software tools that have not no documentation for their developmental process.

Due to most of the software tools that have been built or are in development by RUBi being web applications, the Apache folder was the first to be placed on a repository. Once the folder which has all the web configuration files has been archived, the software of the web applications that are currently hosted was also archived. Lastly, this was followed by archiving repositories of the software tools which are in development. All these software tools are located on the RUBi webserver called *Padme*, which is the server that hosts the various web applications. However, before this archiving process can be undertaken the, the *Git* software package had to be installed onto *Padme* from Ubuntu 20.04 LST package manager using the *sudo apt install git* command on the terminal. Once the *Git* software was installed, the archiving process mentioned above could then be carried out (figure 2.4).

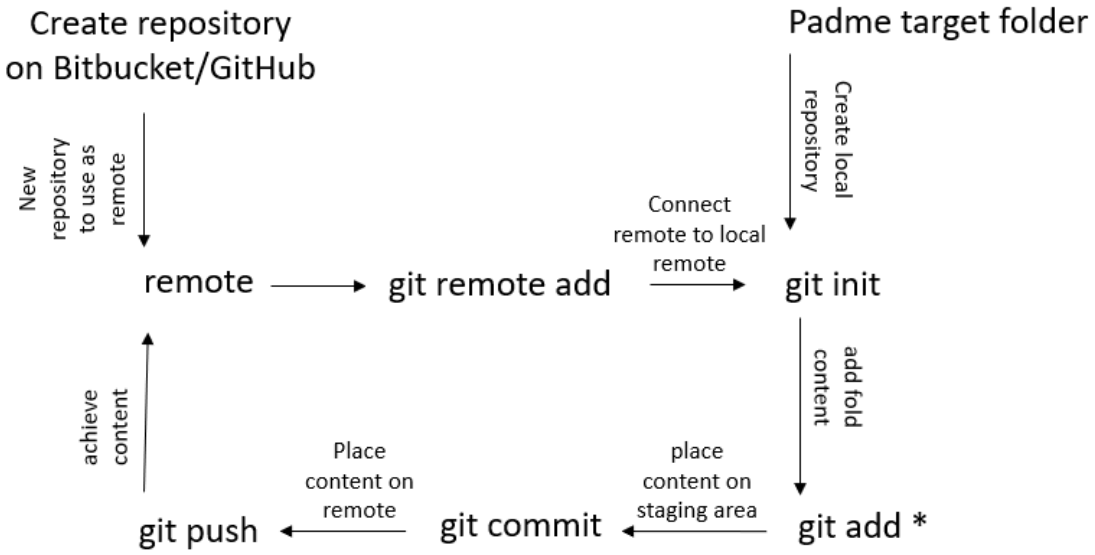

Figure 2.4: Archiving process of software tools and target directories on Padme server

The process followed by using VCS platforms to create repositories that are then used as remote to archive the various software tools developed by RUBi and directories that serve an important function.

Figure 2.4 shows the *Git* commands that were used to archive the various software tools and important directories such as Apache configuration file directories. This process was followed for all the software tools used by RUBi stored on the *Padme* server (figure 2.4). The exception is software tools that are still in development, whereby a single repository was created for such software tools, and all those software tools were placed in that repository. That repository on the VCS platform will show all the software tools that are still in development. However, this does not mean that other software tools that are not in that repository cannot be developed in their respective repositories.

Lastly, all the environment and dependencies were also archived. Due to the predominance of web application development in RUBi, web applications use resources that are used by the code. This includes the Python container environment used to develop the software and resources such as images, databases, and others used by the various web applications. Conventionally, VCS platforms are used for software development and in this instance, resources that are also used by the software were archived.

2.2.2 Results

Using GitHub/Bitbucket VCS platforms enabled the archiving of the various RUBi software tools along with functionally important directories. Furthermore, once any changes are made to a particular software tool, the repository which holds that particular software is updated. The continuous update of the archives will give future RUBi developers the ability to trace the evolution (different versions) of the software developed in RUBi and access to the latest versions of that software. Additionally, suppose any fault that results in data loss on the RUBi *Padme* server that stores the software tools. In that case, this redundancy of also having the archives that is up to date can be used to retrieve the software (Saadoon et al., 2021). With the resources used by the software also being present in the repository, all the software requirements are met, and it can be deployed.

Furthermore, the software being developed by RUBi is primarily web-based. The availability of the Apache configuration files means deploying the web applications would just require the Apache software to be installed. This gives RUBi the versatility to migrate its whole web server and web applications to a different web server by simply retrieving those archives, making it a redundant and robust system. Lastly, this also ensures that if a server related failure occurs, the will be minimal web client inconvenience as all the resources are available to ensure all the web applications can be made accessible. However, to ensure this is the case, the repositories need to be continuously updated to ensure the latest version has been archived. This is particularly important for repositories that have databases that are being continuously modified by web clients, as failure to update these databases on the repository will result in some users losing their data as the database version on the repository will be older.

RUBi develops and stores its software locally, resulting in finished software projects with no traceable software development history. Therefore the use of repositories offers a traceable software development history, enabling the flexibility, for example, in using old features that were discarded during the developmental phase to be retrieved and be developed further (Zolkifli et al., 2018; Kubilius, 2014). Additionally, *Git* commands such as the *git branch* can be used to create branches that can be used for adding new features and testing locally. Once the testing of these new features has been concluded, the *git merge* command can be used to publish those features to the branch that is in deployment. This offers RUBi developers a change in their typical software developmental pattern to a more mainstream fashion of software development practices.

It is worth noting that software documentation is also required to ensure that a detailed and in-depth document is available to future RUBi developers about the developed software. This task may prove challenging due to the legacy of the lack of traceability in all spheres of RUBi software development and distributed system structure. However, this is a task of importance and will be of great value to RUBi once it has been successfully drafted. These repository platforms were also used during the maintenance and development of various software in RUBi, which will be discussed later in this thesis.

Chapter 3

Website development and maintenance

3.1 COVIDRUG website development

The use of the web in an increasingly digitising world has made the web an important tool, where content on websites can be used in disseminating information and making available information to a broader audience in the form of web clients (MacFarlane & Bultitude, 2012). This makes websites very valuable aspects of institutions and collectives due to the accessibility to the web, and the mentioned broader audience (MacFarlane & Bultitude, 2012). Websites offer institutions and collectives a platform which can enable communication to be transmitted to interested web clients without directly communicating with members of those given institutions and collective (MacFarlane & Bultitude, 2012).

The content provided on the website can be used as an introduction to the institution or collective (MacFarlane & Bultitude, 2012). Furthermore, the website can serve as the first point of contact with the institution or collective as some web clients would require a certain level of understanding about the institution or collective before making official contact (MacFarlane & Bultitude, 2012). This shows how websites can also function as filters, as interested individuals can use the website to determine whether official contact with the institution or collective is necessary (MacFarlane & Bultitude, 2012). Furthermore, for some institutions or collectives, their websites are not just sources of information for interested web clients but function as a source for grant opportunities.

The use of websites for grant opportunities is also another important reason why an institution or collective would want to develop a website (Smits & Denis, 2014). As potential funders may want to see a website that entails the operation of the institution or collective, as an interested web client, giving the potential funder information on whether the institution or

collective aligns with the funding intention (Smits & Denis, 2014). All these aspects mentioned above show the value having a website can provide to institutions and collectives.

For a website to yield its potential value, there are some attributes that the construction of the website requires (Garett, Chiu, Zhang, & Young, 2016). The intention of having these attributes is to increase the attractiveness of the given website along with increasing its usability, amongst other things (Garett et al., 2016). Some of these attributes are how a web client navigates the website, how the web pages are organised, how the content on the web page is graphically represented, and the readability of the content (Garett et al., 2016). The previously mentioned can ensure that when web clients access the website, they are able to get the information they want while sustaining the web client interest in using the website further (Garett et al., 2016).

The Covidrug-Africa Consortium (COVIDRUG) was formed as a collaborative effort to contribute to the study and respond to the Covid-19 in the African context (see covidrug- africa-consortium.rubi.ru.ac.za). Covid-19 is a global pandemic that has placed the globe in a state of uncertainty, making consortiums like COVIDRUG valuable (Ciotti et al., 2019). The construction of a website for this consortium can yield all the benefits mentioned previously.

3.2 RUBi website maintenance

The RUBi website provides web clients with various information about the research unit (see rubi.ru.ac.za). As developments occur in the research unit, the information provided on the website needs to be updated. Therefore the website requires maintenance on the various web pages contained on the website. Furthermore, the RUBi website also enables users to log onto the website, which results in added content being provided to those users. It is worth noting that such a function would require the RUBi website to be categorised as a web application. Firstly, as previously stated, RUBi uses the Django web development framework for its software development. As mentioned above, the use of frameworks results in an overlap between an interactive website and a web application. Secondly, as the majority of the actions carried out in this section being front end related, the processes used are more aligned with website maintenance than web application maintenance.

3.3 Methodology

3.3.1 COVIDRUG website implementation

The Django web development framework was used to build the COVIDRUG website with the predominant use of the front-end aspect of this framework. Like a typical website, the consortium website contains different web pages. Each page gives various important information about the different aspects of the consortium members, their respective students, and how the consortium collaborates.

The first step was to determine the website's structure in regards to the organisation of the web pages and the contents of those web pages. As previously mentioned, the organisation of the website determines whether the web client can navigate the website and retrieve the information they desire (Garett et al., 2016). Therefore to implement this, a design structure was formulated (see figure 3.1).

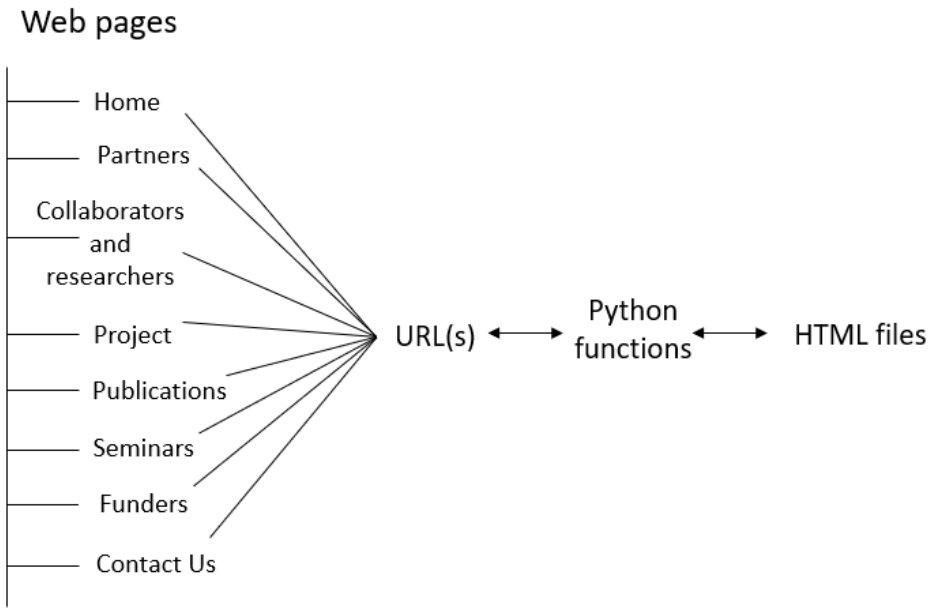

Figure 3.1: COVIDRUG website structure design

The design structure of the COVIDRUG website indicates the needed web pages, the URL(s) that are required, and the Django Python function used to connect those URLs to the appropriate HTML template files.

Figure 3.1, shows all the components that will be developed in the creation of the COVID website using the Django framework. The development of the COVIDRUG website was done on the Padme RUBi server. The web pages will be what the web client interacts with. The URL(s) connects the web pages to the Python functions, which return the HTML template content which is visible on the web pages (see figure 3.1). To simplify the COVIDRUG website's navigation, an HTML file name *master.html* was created using the HTML section tag, whereby the HTML section tag will create a section within the webpage. Additionally, that section was populated by an HTML unordered list tag that was used to give links to all the web pages available on the COVIDRUG website.

Django has the functionality to include and merge different HTML files into a single web page. The functionality is provided in the form of built-in tags which were used to include the master HTML file in all the web pages. For the links in the master HTML file to function, the Python *url.py* script in the COVIDRUG website directory was used to connect these links to their relevant HTML files. This was done in the Django framework convention of creating Python functions in the *views.py* file, which are called in the *url.py* file when a particular web page is required and return the HTML files that provide the content of that web page. This was followed by creating HTML files for all the web pages and populating them with relevant information based on the name of the web page (see figure 3.1). This content in the HTML files includes images, links to websites containing relevant COVID-19 information, and information about the consortium. The graphical representation regarding decorations was done using CSS that was embedded in the HTML files.

Once all the development of the web pages was done, the Django *setting.py* script was populated with directories of all the files that will be used and the *DEBUG* variable was set to true. To determine if the content of the web pages was successfully implemented, the website was tested locally on the Padme RUBi server by executing *python manage.py runserver PadmeIP* on the Padme terminal. The command creates a local instance of the COVIDRUG website, which was accessed to observe how the contents of the web pages are presented. It is worth noting that for the creation of the local instance, the *ALLOW_HOSTS* list variable in the *setting.py* script must contain the IP address of Padme server.

To publish the website to the web, an Apache configuration file was created for the COVIDRUG website. As per Apache directories, the file was created in the *site-available* directory. The Apache configuration file includes various directives and tags that are used by the Apache software to carry its publishing of a website function to the web. The *VirtualHost* tag contains the *ServerName* directive, whereby the URL of the COVIDRUG website was placed, and the *Redirect* directive that has the HTTP link to the COVIDRUG website.

Lastly, the Apache *IfModule* tag contains all the directives used by the Apache software. These include the directives like the *ErrorLogs* that contains the directory of where the error logs of the COVIDRUG website should be placed, while the *SSLCertificate** directives give the Secure Sockets Layer (SSL) certificates information used by the website. On the other hand, the *Alias* directives, which will give the location of the resources used by the website and its content, and the *WSGIDaemonProcess* directive allocates the resource and attributes used to host the website.

Once the Apache configuration file has been populated with the correct directives for the COVIDRUG website, the *DEBUG* variable was set to false, and the configuration was enabled using the *sudo a2ensite file_name* commands on the Padme server. This was followed by restarting Apache using the *sudo service apache2 restart* command and testing whether the COVIDRUG website is accessible on the web. A repository for COVIDRUG website was made, containing all resources and code used for the COVIDRUG website. *Git* commands were used for the creation of the local repository, adding all the changes and staging those changes. Lastly, the local repository was connected to the remote previously mentioned, and the staged changes were pushed to the repository.

3.3.2 RUBi website maintenance procedures

The different functions offered by the RUBi website is discussed, along with the various updates required to update information and resources. Firstly, to achieve such maintenance, understanding the different locations where the web pages and their associated files are stored is important. Since the Django framework was used to develop the website, the design structure of the framework was used to locate the different web pages and their associated files. For productivity, an overview of where these files are found was done (figure 3.2).

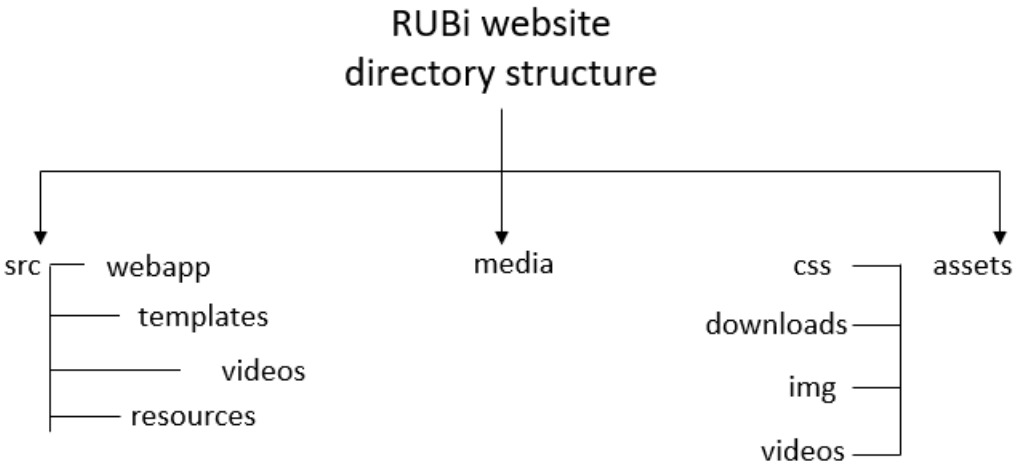

Figure 3.2: The structure of the RUBi website directories that contain content used during maintenance

All the directories containing all the files will need to be accessed and modified for the various maintenance requirement procedures for the RUBi website.

Figure 3.2, shows the mapping of the directories that contain files that will need to be accessed and modified in order to carry out the required maintenance. The *templates* sub-directory as mentioned previously will contain all the HTML files along with its sub-directory *videos* and the *resources* sub-directory will contain the documents that can be used in the front end (figure 3.2). The *webapp* directory will contain the previously mentioned sub-directories along with the Python scripts (figure 3.2). The *media* directory stores the documents uploaded onto the RUBi website. Lastly, the *assets* directory is used as the Django *static* directory and also has the *download* sub-directory, which is used to store files which can be downloaded by RUBi website web clients (figure 3.2). The second step was to draft an overview of the interactions between the RUBi website database and its various web client (figure 3.3).

The RUBi website uses an SQLite database as its back end to store various user information (figure 3.3). This SQLite database is accessed using the Django Admin interface, whereby the database and all the related functions on the website that rely on the database are managed (Django Developers, 2022). These figures show all the parts that will be part of the maintenance of the RUBi website.

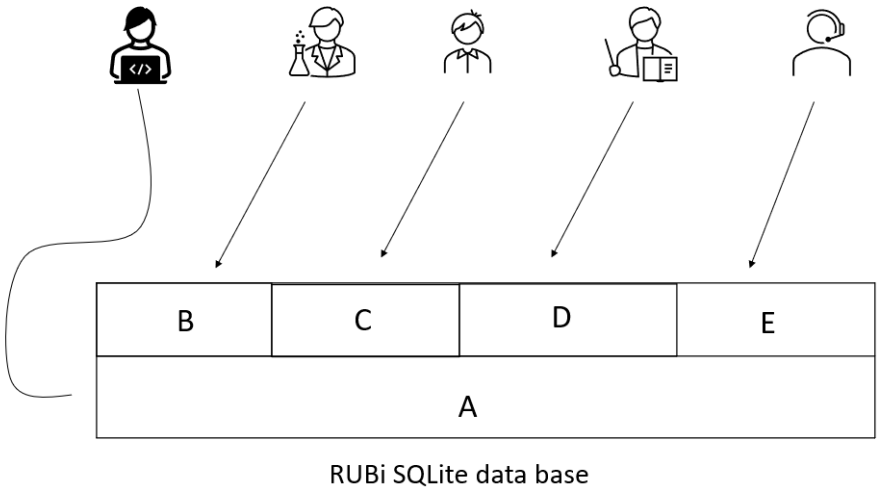

Figure 3.3: RUBi website back end data base access categories

This is a visual demonstration of how different users who have signed up on the RUBi website will interact with the back end RUBi SQLite database.

A: The web administrator will have access to all the tables and there attributes in the database.

B: The members of RUBi can be given access to items such as tutorials which are available on RUBi along with their personal user profiles.

C: The prospective masters student or non RUBi member has access to their user profile and to the application form submission.

D: The lecturers has access to the users profiles and the submitted applications.

E: The lab manager can be given access similar to the web administer but without being able to edit the database.

Due to only the deployed version of the RUBi website being available, in order to carry out the maintenance, a copy was made and stored in the development directory previously mentioned. This was done using the Linux copy command. Once the copy is available, the maintenance can then be carried out on that copy. The testing of the maintenance implementation was done locally, and then the changes were copied back to the deployment directory of the RUBi website. The maintenance that had to be carried out was updating of RUBi member information, document and web page updating, package maintenance and addition of new web pages.

The implementation of updating member information involved two sub-directories, namely, the *templates* and *img* sub-directory (figure 3.2). The implementation involved add images of the members which where stored in the *img* directory and adding the relevant member information in the appropriate HTML files, namely, *students.html*, *academic.html*, *graduates.html* and *support.html*. The updating of the documents included a few instances whereby various

documents had to be updated. These instances of updating the MSc course document offered by the RUBi, the acceptance and rejection documents sent to prospective RUBi students. This was done by adding the mentioned documents in the *resources* sub-directory and changing the *src* HTML tag in the *index.html* in the *templates* sub-directory for the MSc course (figure 3.2). In the case of the acceptance and rejection documents, these are used in the Python code, and the changes were done in the *views.py* script in the *webapp* directory (figure 3.2).

The updating of web pages was done in the HTML files in the *templates* sub-directory. The updates were to add newly published papers by the research unit, adding new images, announcements from the research unit and applications. These updates were done on the *publications.html* web page for the newly published papers, *index.html* web page for adding images and announcements, the *apply.html* web page for applications. As mentioned the images were place in the *img* sub-directory and the HTML files are in the *templates* (figure 3.2). Furthermore, the updating of the *apply.html* web page for applications involved the changing of the RUBi website database (figure 3.3). The apply date field to ensure the application is visible at the chosen date so prospective RUBi students can apply (figure 3.3).

Package maintenance was done on the RUBi website for LibreOffice which is used for document interaction. This was due to the failure of the LibreOffice software to convert RUBi website documents to a Portable Document Format (PDF). Upon investigation, which was done by locating the file carrying out this functionality and manually reviewing the packages being used in the process of generating the PDF. It was found that the software package installed on Padme used by the RUBi website was broken. This was repaired by first uninstalling the broken LibreOffice and all its dependencies, then installing a new version (figure 3.4).

```
sudo apt-get purge libreoffice?
sudo zypper rm --clean-deps libreoffice*

wget -b https://www.libreoffice.org/donate/dl/deb-x86_64/7.3.2/en-GB/LibreOffice_<Version>_Linux_x86-64_deb.tar.gz
tar zxvf LibreOffice_<Version>_Linux_x86-deb.tar.gz

cd LibreOffice_<Version>_Linux_x86-deb/
cd DEBS/

sudo dpkg -i *.deb
```

Figure 3.4: Commands used to repair the broken LibreOffice software for the usage in the RUBi website

The commands were executed on the Padme server terminal to repair the LibreOffice software. Firstly, the existing LibreOffice and its dependencies were removed, followed by installing a new version of LibreOffice.

Figure 3.4 shows the commands that were used to carry out the package maintenance. As mentioned, this software tool is used by a Python function in the *views.py* script found in the *webapp* directory when interaction with the applications are done by the *apply.html* web page found in the *templates* sub-directory (figure 3.2). Lastly, the addition of new web pages required, firstly, the creation of URLs to access those web pages.

Due to the usage of the Django framework, the *ur.py* Python script in the *webapp* directory. Therefore the Django URL directives were added to the script for the new web pages. Once the URL directives were added, the HTML pages were designed and stored in the *videos* sub-directory of the *templates* sub-directory. With the completion of each phase of maintenance, the Apache webserver had to be restarted using the *sudo service apache2 restart* command on the terminal.

3.4 Results

3.4.1 COVIDRUG website

The website's design was solely done using HTML tags and Django built-in reference tags (front end), enabling a simple design for quick development. However, for any website to have any web client interaction or even sustain a web client interest, there are web design features that, when implemented, achieve this (Garett et al., 2016). The creation of the structure gives an overview of what web pages were needed and, based on the name of the web page, also informs what the content in that would be (see figure 3.1). Furthermore, it also gave all the needed interaction to render the content of the given web pages.

Creating the links into a single master HTML file and using the Django built-in tag to place the master HTML content at the top of each web page enables the web client to have access to all the web pages using those links. Furthermore, this improves the usability of the COVIDRUG websites as this enables ease of navigation. The index web page of the website labelled as the *home* web page contains introductory information about the consortium. This enables the first point of interaction with the web client being a web page that briefly informs the web client about the consortium.

Following the *home* web page is the *partners* web page, which gives further detail about the collectives who are the different researcher participants in the consortium with their images. This will allow the web client to have a brief understanding of the individuals which can inform

the web client which individual they may want to contact and the background of the consortium members (Garett et al., 2016). The *Collaborators and researchers* web page is an acknowledgement page that has all the individuals who are participating in the project pipeline. Once again, this web page gives details about which individuals in the consortium carry out which part of the pipeline. The *Project* web page contains the pipeline used to achieve the goals of the consortium, with the *Publications* web page showing all the published academic work done by the consortium. The *Seminars* web pages contain the public engagements the consortium has had on various platforms. Lastly, the *Funders* web page will show the institution that funds the consortium with the *Contact us* web page contact details.

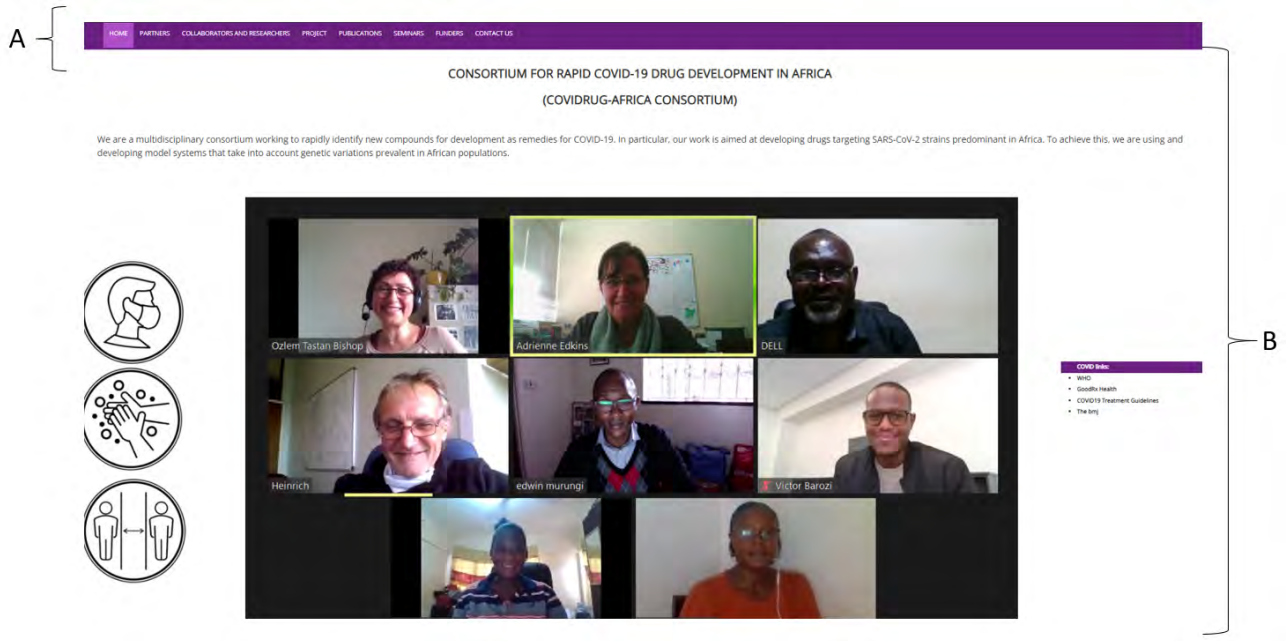

Figure 3.5: Home page design of the COVIDRUG website

The figure shows the current home page design of the Covidrug-Africa Consortium (COVIDRUG) website. Furthermore, it illustrates the general design of the COVIDRUG website. To further study the website, visit covidrug-africaconsortium.rubi.ru.ac.za.

A: The navigation header, with each header having a corresponding web page.

B: The body of the web page with content that is relevant to its navigation header.

Figure 3.5 shows an example of how web pages in the COVIDRUG web pages are constructed. Where the top part of each web page is the embedded master HTML file and the bottom part is the part of the web page that contains content relating to that particular web page (see figure

3.5). CSS template that are embedded in the HTML of the web pages is of a consistent colour throughout the web pages. Due to the spread of "misinformation" on the web and recently in regards to Covid-19, the website imploys impartiality by making links to websites with credible Covid-19 information available (Laato, Islam, Islam, & Whelan, 2020). This ensures that any web client that accesses the consortium website can get further information about Covid-19, showing the importance of the contribution the consortium seeks to make.

The testing of the website locally ensured that each web page was correctly designed and informed which changes needed to be made to ensure all the content was as desired. The *DEBUG* variable being true would show which component of the website failed and thus improve productivity since the sources of the failures are reported. The testing being hosted on Padme requires the IP address to be added to the *ALLOW_HOSTS* list to ensure the Django framework allows the website to be hosted on Padme.

The creation and population of the Apache configuration file ensured that all the requirements that are needed by the Apache software to host the website are available. Django advises that the *DEBUG* variable is set to false when the website is deployed into production (Django Developers, 2022). The setting of the *DEBUG* variable to false ensures that when the website fails, it fails gracefully without pushing the error reporting results to the front end (Django Developers, 2022).

The *a2ensite* Apache command will publish the website and make it accessible on the web. Once the COVIDRUG website is enabled, the Apache webserver needs to be restarted to pull any changes found in the configurations. The checking of the accessibility indicated whether the website could be accessed on the web and if its attributes were rendered. Placing the website on a repository ensures furthermore development of the website can be made using a VCS, yielding all the benefits of using a VCS (Zolkifli et al., 2018; Kubilius, 2014).

The website will give the consortium a broader reach to various web clients and can even be made available to possible funders to ensure the consortium can effectively contribute to the battle with Covid-19.

3.4.2 RUBi website

The drafting of all the directories that will be used in the various maintenance implementation procedures was important to have an understanding of all the areas of interest (figure 3.2). Additionally, if an error were to occur during the testing, an idea of where the source of the error may be located is offered by such a draft. Furthermore, this also offers the comprehension of the other areas which may be affected by the application of the maintenance. Lastly, the drafting of the database interaction between the RUBi website with the different web clients also provides comprehension of the maintenance that has database-related implications (figure 3.3).

Making a copy for testing, particularly in software development instances like the application of maintenance procedures, ensures that the deployed RUBi website is not affected if a mistake is made. Furthermore, testing can be done on the copy version of the RUBi website to ensure all the maintenance has been done successfully and no other parts of the website have been affected. This approach is not the most optimal. A more optimal approach would be using a VCS.

Due to the legacy of web development in RUBi, the application of the maintenance was not done using a VCS. The reason is that historically maintenance was done locally without the use of a VCS, so this practice was carried over into this maintenance as well. However, using a VCS would give the ability to create testing or development branches and a master branch for the deployed version of the website within the same repository (Zolkifli et al., 2018; Kubilius, 2014). As mentioned, due to the RUBi legacy practices, the testing and production versions were put into separate directories on Padme to address this particular legacy issue. It is advised that in the future, the RUBi website repository, which has been created to establish the mentioned branches where the branches feature of VCS can be used optimally (Zolkifli et al., 2018; Kubilius, 2014). The branching feature will save local storage space on the Padme server that hosts the website due to only having a single copy of the website (Zolkifli et al., 2018; Kubilius, 2014). Furthermore, this will also enable the merging and reverting between different versions and branches along with the benefits mentioned previously of using a VCS (Zolkifli et al., 2018; Kubilius, 2014).

The successful implementation of the maintenance enabled the RUBi website to have relevant and up to date content. This ensures that web clients interested in the research unit have all the details they may need. This includes a web page that has all the published papers, the staff and the students in the research unit. The addition of the web pages was done to host

research information about one of the students of the research unit.

The addition of the new web pages into separate locations was done because the HTML content is not directly related to the RUBi website generally. Thus the new *videos* sub-directory was created because these HTML web pages were just hosted there for accessibility. The repair of the broken LibreOffice enabled the RUBi website to send email notifications to prospective RUBi students regarding the response from the research unit regarding their applications. The restarting of the Apache webserver software enables the changes that have been made through the maintenance procedures to be reflected on the live RUBi website.

Chapter 4

Web application enhancements

As mentioned previously, web application maintenance forms a vital part of the life cycle of a web application (Tian et al., 2021; Mohan & Greer, 2018). For any producers of web applications, a maintenance plan of their web applications should be part of the long term objective once those particular web applications are in deployment (Tian et al., 2021; Mohan & Greer, 2018). This long term plan can ensure that when maintenance is done, sufficient resources and planning have been made available to carry out the maintenance with minimal web client inconvenience (Tian et al., 2021; Mohan & Greer, 2018). Websites are typically composed only of mainly front-end components. A web application, on the other hand, contains front-end components and also back-end components (Prokhorenko et al., 2016). This added back-end feature of web application tends to give more points of degradation and requires more extensive maintenance (Prokhorenko et al., 2016).

The back-end components are continuously changing to adapt to various factors, such as applying security measures and also remaining competitive with other techniques which can be used to achieve the same outcome (Prokhorenko et al., 2016). From this perspective, various aspects need to be observed from the point of view of web application maintenance. This includes the change in web client requirements and the evolution of different technologies used by the web application, particularly the ones in deployment (Prokhorenko et al., 2016). These are the two general aspects amongst other localised factors that should inform the maintenance procedure, and deadline (Tian et al., 2021; Mohan & Greer, 2018).

The web client requirements is having an understanding of how the web application can be improved to facilitate ease of use of the web application by web clients (Tian et al., 2021; Mohan & Greer, 2018). Furthermore, the needs of a web client to use a web application optimally changes over time, based on the type of web application and the target end-user (Tian et al., 2021; Mohan & Greer, 2018). Therefore, consistently encouraging web clients who use the web application regarding their desired functionality needs on the web application is important to inform what web clients require (Tian et al., 2021; Mohan & Greer, 2018). On the other hand, observing the evolution of web application technologies gives producers of web application ideas of how their respective web applications can be improved to keep up with other similar updated or competing web applications (Tian et al., 2021; Mohan & Greer, 2018). Additionally, the observation can also show what changes need to be made in regards to the integrity of the web application, and the sustainability of the backend technologies (Tian et al., 2021; Mohan & Greer, 2018).

RUBi develops various web applications primarily for biological applications purposes and other web applications used daily for interaction with the RUBi distributed server system. One of the biological analysis web applications is MDM-Task web (mdmtaskweb.rubi.ru.ac.za), while web applications like Job Management System (JMS) can be used to manage the RUBi distributed server system.

4.1 MDM-Task Web

MDM-Task Web is a web application that hosts MD-Task and MODE-Task software tools (Amamuddy, Glenister, Tshabalala, & Bishop, 2021). However, new additions to the web application were developed, and thus a maintenance protocol had to be made. Due to the academic nature of the MDM-Task Web, the maintenance of this particular web application comes in the form of adding functions that MDM-Task Web reviewers advised will be advantageous to web clients, improving user experience and convenience.

Additional changes will have to be made to accommodate the adjustments and adapt various development techniques that will enable the changes to be implemented successfully. All of this will form part of the implementation process to achieve the set maintenance requirements. Briefly, the maintenance includes extending the functionality of selecting previously used trajectory files to also apply to topology files and adding new protein visual presentations. Trajectory files provide a simulated molecular system in a text file format that represents atomic

coordinates at a given time, while topology files show the atoms are connected to each other through chemical bonds (Salomon-Ferrer, Case, & Walker, 2013).

4.1.1 Methodology

The implementation process was done using GitHub as the version control system (VCS). A private GitHub repository was created, followed by the creation of a local repository in the directory which had the MDM-Task Web resources. This directory was then pushed as the first version to the repository. The first implementation approach was to understand how front-end components interact with back-end components.

MDM-Task Web is composed of four structural components which define its functional aspect (figure 4.1). Furthermore, the additions affect different parts of those structural components, depending on the type of addition. However, these structural components are all within the Django web development framework. It is worth noting that the jobs submitted by the web clients of MDM-Task are executed on a different RUBi compute server. This is done by sending a request from MDM-Task that is hosted on Padme to the Job Management System (JMS) site (jms.rubi.ru.ac.za/) that will then execute the job to the RUBi compute server Chewbacca. Therefore, the execution of jobs in the MDM-Task is executed externally, but all the components that carry out the different functionalities are within the MDM-Task directory.

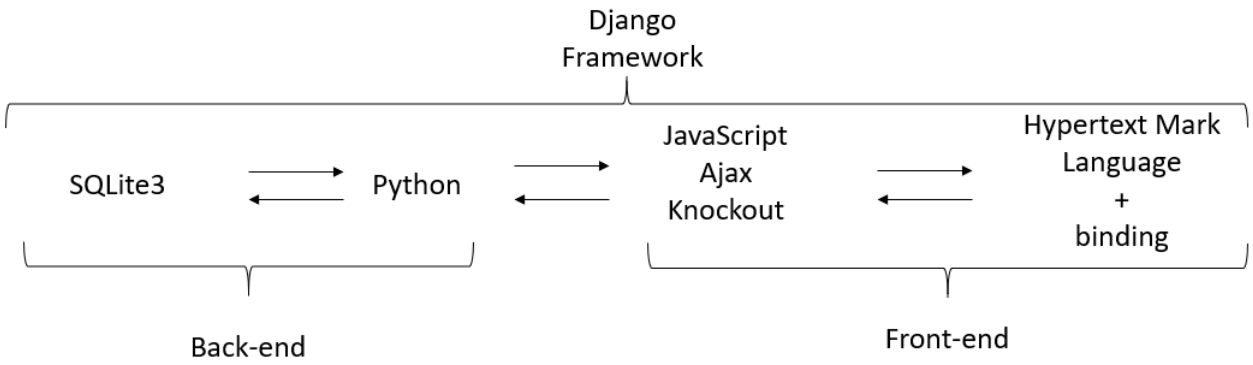

Figure 4.1: MDM-Task Web structural components interactions

These four structural components are how MDM-Task Web components carry out their functional aspects to respond to various web client requests. The response to a particular web client request may not use all the components.

Figure 4.1 shows all the different components, but as stated, the execution of the jobs is not included in the diagram. The HTML will be the part with which the web clients interact, and some of the HTML tags used in MDM-Task Web will have the Knockout Library binding attributes (figure 4.1). This binding will determine the type of rendering in that HTML tag. However, some HTML tags use JavaScript to give MDM-Task interactivity, while Ajax is used to send web client requests to back-end components. Ajax sends web client requests by exploiting the Django web development framework and URL system. The Ajax URL feature will enable the JavaScript code to call Python functions that respond to the web client request. The SQLite3 database is used to store user access information using Django Python libraries. Once the functional components are drafted, as briefly summarised, the first implementation of extending the functionality of the uploading trajectory file feature to topology files for all web pages with the topology file usage feature started (figure 4.2)

Figure 4.2: MDM-Task Web trajectory file input features

This figure is a screenshot of how the input trajectory feature's front-end components (HTML and CSS) are implemented in MDM-Task Web.

A: The HMTL radio button will enable the upload button to open the web client local machine file manager.

B: The radio button will enable the select button that will give a drop-down containing previously uploaded trajectory files.

Figure 4.2 shows the frontend implementation of the usage of the trajectory files in MDM-Task and servers to indicate how this implementation should be extended for topology files. Due to the structural similarities between the frontend requirements of the topology files and the implemented trajectory files, the trajectory implementation was copied to the topology function with the appropriate renaming. However, due to the local storage requirement of the feature, storage changes had to be made to accommodate the functionality.

For usage of previously uploaded files to be implemented, two subdirectories for each web client were created. These subdirectories were to store the topology and trajectory files. This was followed by the implementation of a checking system which determines whether the web client has the two subdirectories. If the web client does not have the subdirectories, the directories will be created. The topology frontend components such as HTML, CSS, and JavaScript were copied from the trajectory frontend functionality. For the backend Python component, the Python OS library path exist functionality was then used for the check system implemented by a Python function called *create_topNtraj* (figure 4.3).

```
def create_topNtraj(job_type):
    if not (os.path.exists("{}/{}".format(settings.DATA_DIR, job_type))):
        os.makedirs("{}/{}".format(settings.DATA_DIR, job_type))
    if not (os.path.exists("{}/{}/topologies".format(settings.DATA_DIR, job_type))):
        os.makedirs("{}/{}/topologies".format(settings.DATA_DIR, job_type))
    if not (os.path.exists("{}/{}/trajectories".format(settings.DATA_DIR, job_type))):
        os.makedirs("{}/{}/trajectories".format(settings.DATA_DIR, job_type))
```

Figure 4.3: Implementation subdirectory checking system for the topology and trajectory files storage

The use of the Python OS library to implement a checking system that checks whether all web clients have subdirectories to store their files. The subdirectories will be for the storage of the topology and trajectory files, respectively. The checking system will create those directories if they are not found.

Figure 4.3 shows how the checking system was implemented. All functionalities relating to the usage of topology and trajectory files usage call the *create_topNtraj* Python function (figure 4.3). It is worth noting that additional functionality was added such as the *store_upload* function which facilitates the storage of the trajectory and topology files. The validation of the topology and trajectory files was changed in the maintenance whereby the Python *mdtraj* library *load_frame* function was used, replacing the *load* function. Due to the implication of

storing both the trajectory and topology files uploaded by web clients, optimisation in storage usage was also implemented.

Due to the varying sizes in which trajectory files can be generated, the first step was to improve storage usage by imposing a size limit on trajectory files a web client can upload and store. This was done in the frontend by editing a JavaScript section for all web page scripts that handle the uploading request of a trajectory file. The additions to the JavaScript section will check whether the upload request of the trajectory file is not greater than 250 Megabytes. If the upload request is greater than the set file limit, an error will be thrown and an HTML modal will to be used. The second storage usage optimisation that was implemented was in regards to how the parameters used by example jobs are accessed by web clients (figure 4.4).

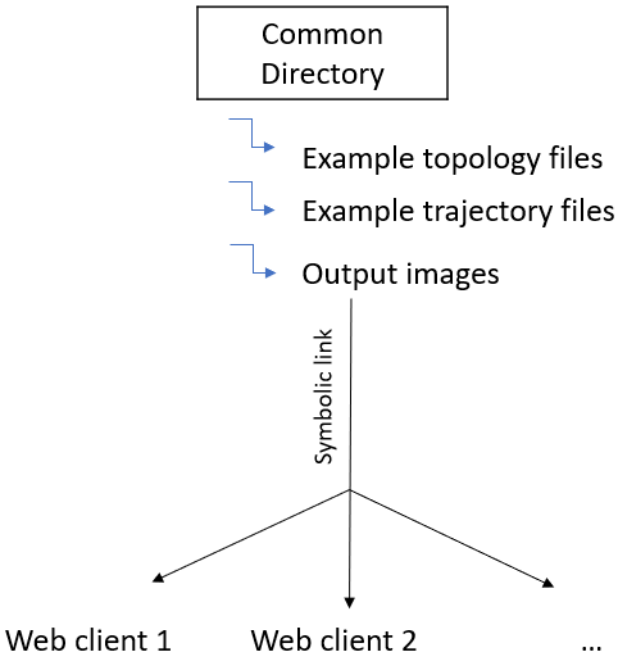

Figure 4.4: MDM-Task Web example and demonstrations parameters structure

The structure of how web clients of MDM-Task Web access the parameters for output examples and demonstrations. The access is given by using the Python OS library *symlink* method. These examples and demonstration parameters are accessed via the View demo and *Example submission* button present on each analysis web page on MDM-Task Web.

Figure 4.4 shows how the *symlink* function from the OS Python library was used to give web clients access to example topology and trajectory files used by requesting an example job for a given MDM-Task Web tool (Lynch, 2018). This functionality was added to the section where a web client signs into MDM-Task Web using the *demo_creat* function that is called when a

web client signs up or logs into MDM-Task Web. A checking system will determine whether that given web client has symlink connections with all the example files needed to request an example job. If a web client does not have the symlinks, the symlinks are created.

Lastly, to improve the offerings of the web application, more molecular visualisation of the protein structure options were added to the current visualisation of using the NGL *cartoon* graphics. The visualisation options were sourced from the NGL viewer, which provides various molecular graphics software packages (Rose et al., 2018). Four visualisation options were selected, the Spacefill, Licorice, and Surface graphics software packages from NGL Viewer (Rose et al., 2018). The NGL software packages were taken in as a library that was in JavaScript format. This required the implementation of these graphics to take place in the JavaScript part of the MDM-Task Web.

For this to be achieved, an identification of the web pages and their respective Java Scripts files that render visualisation was done. The *job.js* script was found to be the JavaScript file that did all the visualisation rendering, followed by adding the graphics software packages to the relevant sections to the *job.js* file. Due to the NGL software graphics package being taken in as a library, it was implemented by calling the appropriate graphics from the library, which will then implement the visualisation (figure 4.5).

```
var cartoon;
cartoon = ol[0].addRepresentation("cartoon", {
    color: schemeId,
    visible: true })

var spacefill;
spacefill = ol[0].addRepresentation("spacefill", {
    color: schemeId,
    visible: false })
var licorice;
licorice = ol[0].addRepresentation("licorice", {
    color: schemeId,
    visible: false })
var surface;
surface = ol[0].addRepresentation("surface", {
    color: schemeId,
    visible: false })
```

Figure 4.5: NGL software graphics package implementation example

The initialisation of the variables that will be used for the various visualisation graphics, namely, *cartoon*, *spacefill*, *licorice* and *surface*. Lastly, this is followed by the calling of the graphics from the NGL software package library with their attributes into the set variables.

Figure 4.5 shows the various graphics from the NGL software package that were implemented for visualisation. This was done by first creating global variables, followed by calling the respective graphics from the NGL Viewer and setting their attributes into the relevant variables (figure 4.5). For the web client to have the ability to select between the new offered graphics, an HTML *radio* button was used as the selector. These HTML *radio* buttons were added at the appropriate sections of the *jobs.html* file, along with the *data-bind* Knockout attributes. The Knockout attributes *data-bind* were then observed in the *job.js* file to determine which HTML *radio* button was activated (figure 4.6).

A

```
<input type="radio" value="cartoon" name="Drawing" value="cartoon" data-bind="checked: Drawing">
<label >Cartoon</label>
<input type="radio" value="spacefill" name="Drawing" value="spacefill" data-bind="checked: Drawing">
<label>Spacefill</label>
<input type="radio" value="licorice" name="Drawing" value="licorice" data-bind="checked: Drawing">
<label>Licorice</label>
<input type="radio" value="surface" name="Drawing" value="surface" data-bind="checked: Drawing">
<label>Surface</label>
```

B

```
self.Drawing.subscribe( function(newValue){
    if(newValue == "cartoon"){
      cartoon.setVisibility(true);
      spacefill.setVisibility(false);
      licorice.setVisibility(false);
      surface.setVisibility(false);
    }
    else if(newValue == "spacefill"){···
    }
    else if (newValue =="licorice"){···
    }
    else if (newValue =="surface"){···
    }
  })
```

Figure 4.6: HTML and Java Script implement of the NGL graphic visualisation selector

A: The use of the HTML *radio* buttons as the selector between the different molecular visualisation graphics that contain Knockout *data-bind* attributes.

B: The JavaScript section that observes the Knockout library *data-bind* attributed to determine which NGL Viewer graphics should be rendered.

Figure 4.6 shows an example of how the previously mentioned selector system was implemented. All the previously mentioned were periodically pushed to the GitHub VCS once it was determined that significant changes had been made to the MDM-Task Web.

4.1.2 Results

The creation of the GitHub repository formed a location where changes made can be stored, enabling the traceability of implementing the maintenance procedure. Furthermore, future maintenance procedures can use this repository for such implementation.

Figure 4.7, was used in the implementation approach to understand which parts of the structural component require changes to accommodate the various additions. This will ensure that the full extent of the changes is done appropriately and understand the possible location of failures during the implementation. Furthermore, figure 4.1 reveals that the structural components are highly dependent, causing coupling amongst the functionalities offered by MDM-Task Web. As previously mentioned, coupled components during the maintenance of any software result in multiple changes being required throughout each component, and multiple changes increase the probability of errors and failures occurring (Tian et al., 2021; Mohan & Greer, 2018).

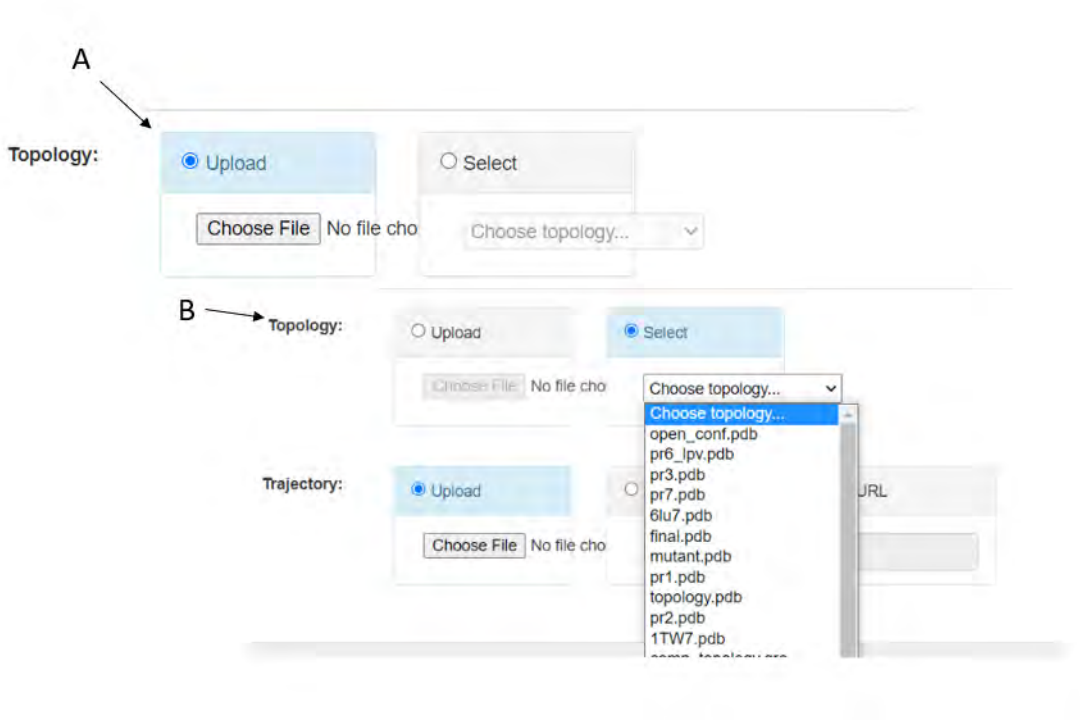

Figure 4.7: Implementation of topology files input types

The implementation of the selection feature of previously uploaded topology for all the web pages that have topology file inputs.

A: The MDM-Task Web functionality to upload topology files from the web client local machine.

B: The MDM-Task Web functionality to select topology files that have been previously uploaded.

Figure 4.7 shows the frontend perspective of how the trajectory input file was extended to topology files. Furthermore, the MDM-Task Web frontend design also has redundant code, such as the same input types being written for each web page of the web application. Therefore, when the functionality is added to the Python part, the JavaScript and HTML components will also need to have additional functionalities (figure 4.1). This results in an increased maintenance period as the additions will need to be extended to all the web pages and also all the different components relating to that web page (figure 4.7).

All the previously mentioned issues, amongst other issues, indicate that implementing the requirements will need to either minimise the full effect of these structural issues or completely solve them. This is to ensure that the maintenance is not only being done to add the advised functionalities but also to use this opportunity to improve the general maintainability of the web application. Firstly, it was done by creating the *create_topNtraj* function that is called by all the Python functions that handle trajectory and topology subdirectory creation (figure 4.3). This optimisation will enable any future change that may be done concerning the subdirectories in question to require a change in only one section of the code (Aleti, Buhnova, Grunske, Koziolek, & Meedeniya, 2012).

Secondly, to facilitate the storing process of the trajectory and topology files while improving maintainability a *store_upload* function was created to store the file in their respective subdirectory. Similarly to the *create_topNtraj* function, the *store_upload* function is called in all sections that deal with the mentioned files. Therefore, it is another point where the maintainability was improved as any future change in regards to the files storage only requires a single section of code to be changed.

Thirdly, implementing the trajectory size limit ensured that the uploaded trajectory files do not burden the RUBi storage systems. Setting the constraint on file size for web clients leaves the possibility of web clients being dissatisfied since some web clients may want to upload larger trajectory files (Al-Maskari & Sanderson, 2010). However, preserving the sustainability of RUBi storage server takes precedence as other aspects of the research unit need storage to carry out different functions. The HTML modal will inform the web client that the file they are uploading is over the size limit (figure 4.8).

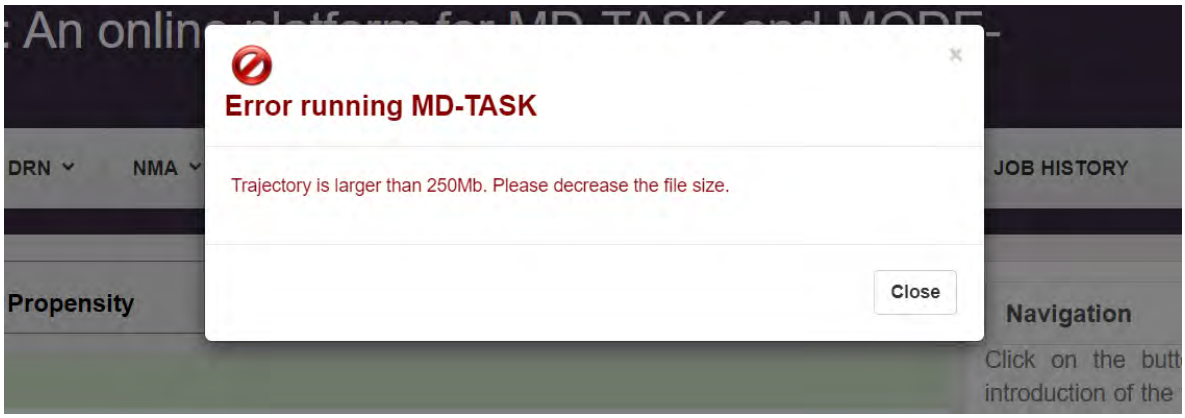

Figure 4.8: Implementation of the HTML trajectory file size error

An example of the HTML modal that will be displayed when a web client attempts to upload a trajectory file that is larger then 250 Megabytes.

Lastly, the function of examples and demonstrations is to show the web client what type of results to expect for each tool offered in MDM-Task Web (Amamuddy et al., 2021). These features are available to all web clients and would mean in terms of how MDM-Task Web is designed that every web client would need a copy of all the required parameters for the examples and demonstrations for each tool in the web application. This entails that if there are many web clients, multiple copies of the examples and demonstration files would need to be made available, which has storage implications. Therefore, the implementation of the *symlink* ensures that all web clients have access to the examples and demonstration content from a single copy. The use of a single copy ensures that when new examples and demonstrations are developed, only a single section of the code needs to be changed to give users access to that new content. This improves both storage utilisation and maintainability.

As previously mentioned, when a web client accesses MDM-Task Web by logging in or signing up, a check is done to determine whether the web client has the examples and demonstration content symlinks using the *demo_creat* function. This check was done to ensure that any web client of MDM-Task Web that previously used the web application before the maintenance took place also has access to the examples and demonstration content via symlinks. However, this feature offered by the *demo_creat* function can be called during the signing up process, but as mentioned, the feature was added after added production phase. Resulting in a need also to call the *demo_creat* function during the logging process.

The addition of new graphics from the NGL Viewer software package gives web clients the ability to visualise their molecular structures in different graphics. Furthermore, the relief from the constraint of only having a single graphic for comparison can yield greater web client satisfaction (Al-Maskari & Sanderson, 2010). As a result, web clients have the option to choose which graphic to use for their comparison (figure 4.9).

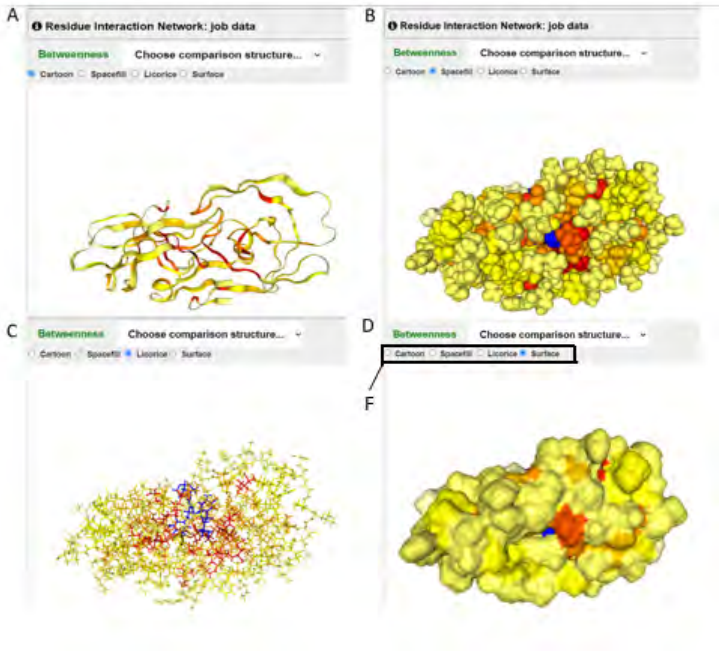

Figure 4.9: MDM-Task Web NGL Viewer residue visualisation options

A: Cartoon view; **B**: Spacefill view; **C**: Licorice view; **D**: Surface view.

F: HTML radio buttons in MDM-Task Web that enable the rendering of different NGL Viewer reside visualisations.

Figure 4.9, shows the different molecular visualisation options offered in MDM-Task Web. The use of an HTML *radio* button to select the graphics ensures that only one graphic can be selected at any given time. All the maintenance carried out on MDM-Task Web mentioned above was to respond to the reviewer's suggestions and further improve the maintainability of the MDM-Task Web and its ecosystem. As mentioned in this section, these came in the form of enhancing the storage capacity and the execution structure to ensure that the future maintainability prospects of MDM-Task Web are grounded.

4.2 Job Management System (JMS)

Job Management System (JMS) is a web-based Workflow Management System for distributed computing servers architecture (Brown, Penkler, Musyoka, & Bishop, 2015). JMS facilitates the monitoring, submission, management and execution of jobs via a web-based interface and is used by RUBi to carry out these functions for its distributed computing servers system (Brown et al., 2015). RUBi uses Torque as its resource manager and job scheduler software on its compute servers. JMS has been configured to execute various Torque commands for the monitoring aspects, running jobs and determining the status of the jobs to update the web interface (Brown et al., 2015). Furthermore, a MySQL database is used to keep track of all the various attributes of the jobs that are running on the compute servers. Similarly, the job submission and management follows the same logic of running Torque commands, with the primary advantage being that users of Torque interact with the scheduler through the JMS web application to achieve all the mentioned functionality instead of the terminal (Brown et al., 2015).

JMS was designed to be an open-source software that runs on a Linux distributed system (Brown et al., 2015). This entails that all the different software components that make up JMS should also be open source, including the resource manager and job scheduler that JMS interacts with to maintain its open-source characterisation. Furthermore, this highlights the previously mentioned need for developers of a given software to remain up to date regarding any information about their open-source software components that they used in development (Venters et al., 2018; Stol & Ali Babar, 2010). In the context of RUBi, this became evident as the open-source Torque resource manager and job scheduler that runs on Ubuntu 16 RUBi servers could no longer be in long term usage. This is due to Ubuntu 16 reaching its end-of-life, thus requiring the need for a new Operating System (OS) to be sourced that is receiving updates and is being maintained.

Furthermore, the Torque resource manager and job scheduler version that is open-source can only run on Ubuntu 16, and any new versions of Torque are only available at a cost for new Ubuntu flavour version. These two previously mentioned factors remove JMS as software that can be categorised as open-source. Any implementation of JSM in its current form will require the purchase of Ubuntu 16 support in the context of RUBi or purchase of Torque support since new versions of the resource manager and job scheduler are not free and not open-source.

Lastly, Python 2.7 was used to develop the whole back-end stack for JMS functionality using the Django framework. However, Python 2.7 was placed under deprecation, which means that there will no longer be any releases for Python 2.7 from added libraries that support new functionality to security patches to improve the security of the various libraries. Therefore the deprecation of Python 2.7 does not only affect the future maintenance prospects of JMS, but also its security integrity (Venters et al., 2018).

To address the previously mentioned issues, the back-end of JMS will undergo maintenance to resolve these issues. The primary objective of the enhancement of JMS was to update the Python version used by the JMS and the resource manager to replace the Torque. Lastly, the maintenance will also include various adjustments that must be implemented to accommodate the maintenance addition.

4.2.1 Methodology

Before any implementation of the maintenance can be established, there first needs to be an identification process that will determine the best replacements for the software tools mentioned above. These software tools in question are the Python version and the resource manager and job scheduler to be implemented on the compute servers. In addition, Ubuntu flavour version 20.04 Long Term Support (LTS) Focal Fossa needs to be sourced as the Operating System (OS) that will be used.

Table 4.1: Different Python versions along with each version information

Python Version	**Release Date**	**Deprecation Date**	**Reported bugs logs**
Python 3.6	23 Dec 2016	23 Dec 2021	54
Python 3.7	27 June 2018	27 Jun 2023	57
Python 3.8	14 Oct 2019	14 Oct 2024	45
Python 3.9	5 Oct 2020	05 Oct 2025	33
Python 3.10	4 Oct 2021	04 Oct 2026	25

The table contains different Python versions, along with when the versions were released and when they will be decommissioned. The *Reported bugs logs* column shows the amount of bugs logs that have been reported for each version. It is worth noting that the details above were sourced from the Python organisation, and the information in the table is subject to change as new development occur.

There are numerous Python versions available in the given category to select the Python versions to be used as a replacement(table 4.1). The consideration was that the version must be reasonably new to ensure a relatively long period before it is deprecated. Additionally, the version must have been sufficiently developed to ensure that the version is stable and has all the needed libraries and package support. For this *Reported bugs logs* which are logs that contain different errors which have been reported for the given version, was used to determine the stability of the version (table 4.1). It is worth noting that the reported bug logs tended to be related to the version release date, whereby older versions would have more reported bug logs due to typically more users using those given versions (table 4.1). So having more reported bug logs is not necessarily an unfavourable attribute for a given version to have.

For the prospects of finding a Python version to use, it was decided that Python version 3.8 best fits the considerations made during the selection although it is set to depreciate in 2024. This was due to the fact that Python 3.8 was the most stable and also the latest version of Python. As previously mentioned using the latest version of a given software may result in bugs that are yet to be resolved and a lack of full implementation of previous functionalities in older versions. Regarding selecting a resource manager, the primary requirement for a resource manager and job scheduler is to be open-source and free. There are numerous mainstream open-source and free resource managers available, such as SLURM and OpenPBS, to name a few.

SLURM is a Simple Linux Utility for Resource Management, a resource management and job scheduler software that is specifically made to run on Linux OS compute servers (SLURM contributors, 2021). Similarly, the open-source and free OpenPBS is designed to run on Linux OS compute servers, with the commercial version of OpenPBS being available for Windows OS compute servers (OpenPBS contributors, 2022). Any of the two previously mentioned resource management and job scheduling software can fulfil the functions required by RUBi. For this reason, the selection was then based on the effort required to set up the mentioned resource management and job scheduling software.

Therefore, OpenPBS resource management and job scheduler was selected as a recommendation from one of the collaborators of the RUBi research lab. The collaborator has an existing system that has implemented OpenPBS as its resource manager and job scheduling software. Therefore, the configuration process of OpenPBS onto the RUBi compute servers during the various implementation processes can be facilitated with the collaborator's assistance. The

two primary requirements for the previously stated objectives have been selected: the Python version 3.8 and the resource manager, which is OpenPBS.

Once the selection process was done, the sourcing of all the selected software and server allocations followed. The server that was allocated was the RUBi compute server *Luke*, which will be used for the implementation of the maintenance. For this to be achieved, the Ubuntu 20.04 LTS Focal Fossa flavour had to be installed on the Luke compute server. The OS ISO image was sourced from the Ubuntu website for the installation process (https://ubuntu.com/download). Once the ISO image was sourced, the Rufus software was used to create a bootable Universal Serial Bus (USB) flash stick containing the Ubuntu 20.04 LTS Focal Fossa flavour OS. The bootable USB flash stick was then connected to the *Luke* compute server and used to install Ubuntu 20.04 LTS OS onto the server. This was followed by updating the OS package manager packages and installing primary packages, such as Ubuntu Build Essentials and others (figure 4.10).

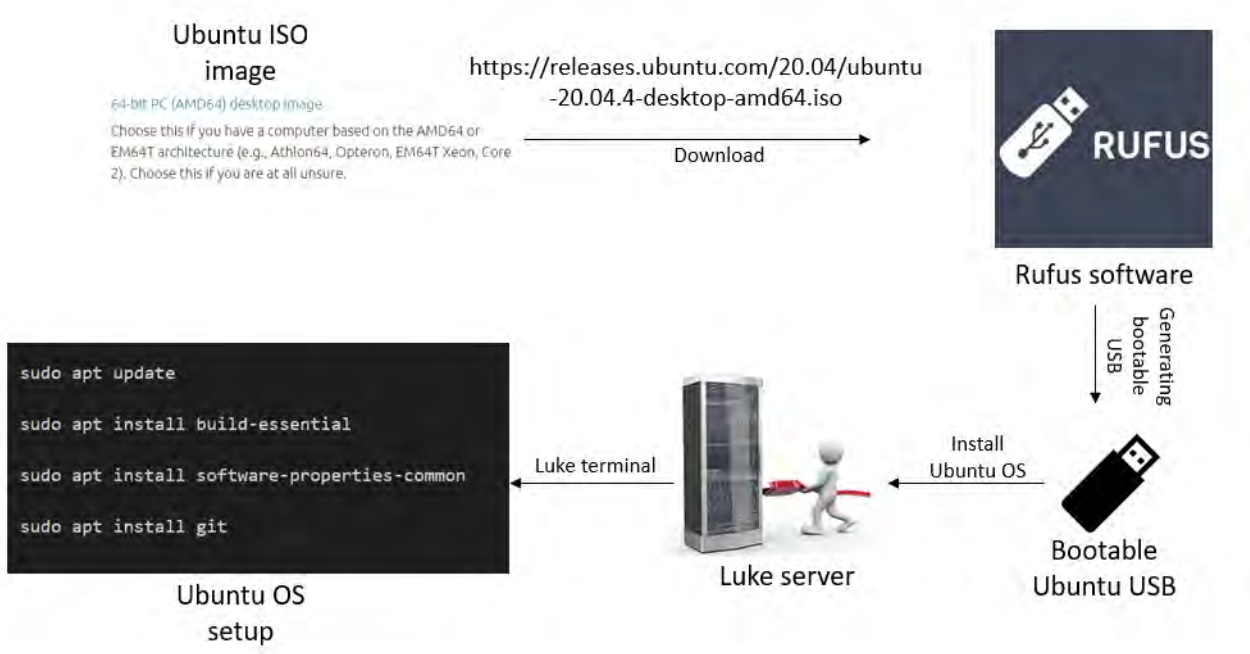

Figure 4.10: Installation process of 20.04 Long Term Support Focal Fossa Ubuntu flavour

The process that was used to source the ISO image of the 20.04 Long Term Support Focal Fossa Ubuntu flavour and the use of the Rufus software tool to create a bootable USB to install Ubuntu on the *Luke* server. Lastly, this was followed update the installed Ubuntu OS on the *Luke* server and installing the *git* software.

Figure 4.10 shows how the installation process that was followed to install the Ubuntu 20.04 LTS OS onto the *Luke* compute server. Once the OS was installed, the sourcing and installation process of the selected software replacements, namely the OpenPBS and Python version 3.8, was undertaken. OpenPBS was sourced from the OpenPBS GitHub repository (https://github.com/openpbs). The OpenPBS GitHub repository offers all the configuration steps to follow in the *INSTALL* document present in the repository. As mentioned previously, a RUBi collaborator assisted in the installation process of OpenPBS on the *Luke* compute server.

Once the installation process of OpenPBS was completed, the testing of OpenPBS was then started. The testing of OpenPBS was done on the terminal of the *Luke* server by first starting the OpenPBS service using the *sudo /etc/init.d/pbs start* command. It is worth noting that the command given in the *INSTALL* document to start the OpenPBS service on a given compute server will only function if all the given installation steps are followed. Following the installation instruction of checking whether OpenPBS commands function, commands such as *qstat -B* were executed, and the output was observed. Furthermore, the *echo "sleep 60"*, *qstat -B* and *pbsnodes -a* command were also executed. Once it was established that all the test commands given in the installation document of OpenPBS gave the stated output, the next step was to source the selected Python version 3.8.

Python 3.8 version was sourced using the Ubuntu 20.04 LTS package manager. This was done by adding the Deadsnake PPA repository as part of the packages available in the Ubuntu 20.04 LTS package manager. The Deadsnake PPA repository contains different builds for different Python versions, which gives the end-user the ability to install a specific Python version. Therefore once this repository was added to the package manager, it was then used to install Python version 3.8. The *Python –version* command was used to determine whether the correct version of Python was installed (figure 4.11). Additionally, the installation of Python version 3.8 was then followed by creation of a local repository, connecting the local repository to a private remote, sourcing of JMS and placing it into the private repository. JMS is available on GitHub (https://github.com/RUBi-ZA/JMS), and the installed *git* package was used to retrieve the JMS. JMS was retrieved by cloning the repository onto the *Luke* compute server at an appropriate location stated in the JMS repository on the compute server (figure 4.11).

```
###### Python version 3.8 install #######

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt install python3.8

sudo apt install python3-venv

sudo apt install python3-pip
```

```
###### JMS sourcing #######

sudo mkdir /srv/JMS

cd /srv/JMS

sudo git init

sudo git remote add origin <JMS-OpenPBS_PrivateRepo>

sudo git clone https://github.com/RUBi-ZA/JMS.git

sudo git push origin master

sudo chown -R userName:userName *
```

Figure 4.11: Commands used for the installation of Python version 3.8 and cloning JMS repository on the Luke server

The commands that were sequentially executed on the Luke compute server terminal.

A: The retrieval of the Python version 3.8 from the Ubuntu 20.04 package manger.

B: The creation of a directory on the Luke server which is then used to store the cloned JMS repository.

Figure 4.11 shows the commands used to retrieve Python version 3.8, the Python packages that will be used and the cloning of the JMS repository into a created directory. Once the Python version 3.8 was sourced, the updating of JMS, which was developed using Python version 2.7, to the system version 3.8 of Python could start. Typical version updates tend not only to come with new libraries and modules but in the instance of programming language software, can include compiler changes resulting in syntax changes. Due to JMS being built using the deprecated Python version 2.7, any updates in Python version 3.8 will have to be implemented. Therefore, all the scripts within JMS were observed to determine whether they needed to be changed to comply with Python version 3.8 specifications.

Various techniques can be used to identify the differences between the Python version 2.7 used to build JMS to the selected Python version 3.8 that will be used as its replacement. A few techniques were considered as an approach path. The first technique was to rely on the Python version 3.8 error detection protocols to pick up any errors during the execution of JMS and then go to the actual scripts and fix those errors. The second technique considered was using a visual inspection of all JMS scripts and determining which scripts will need to be changed, followed by changing those scripts that have been identified. The third technique that was considered was the usage of an Integrated Development Environment (IDE). The IDE can integrate various programming language packages that simplify the developmental processes.

There are different drawbacks to using the techniques mentioned above. For the first technique, the drawback is executing Python for every script in JMS. Given the number of JMS scripts, this technique could be a tedious process. Furthermore, this is coupled with relying on Python error checking to provide accurate error reporting. The second technique drawback is human error-related, whereby some scripts may be missed resulting in such errors only being picked up when executing the scripts. Given the various packages offered by IDEs to assist in most software development activities, it is difficult to determine a drawback in that regard.

However, IDEs are also software tools that can be erroneous. These errors can manifest in various forms that can hinder the maintenance process, such as not picking up all the syntax issues before execution when a given script is opened in the IDE. Therefore, the first technique of executing the scripts and the third technique of using an IDE were used to address the drawbacks mentioned above. The error detection before script execution offered by some IDEs was used to correct errors, while the first technique was used to confirm whether all the errors had been corrected. Figure 4.12 shows a process flow chart that details how these two techniques were used in conjunctions.

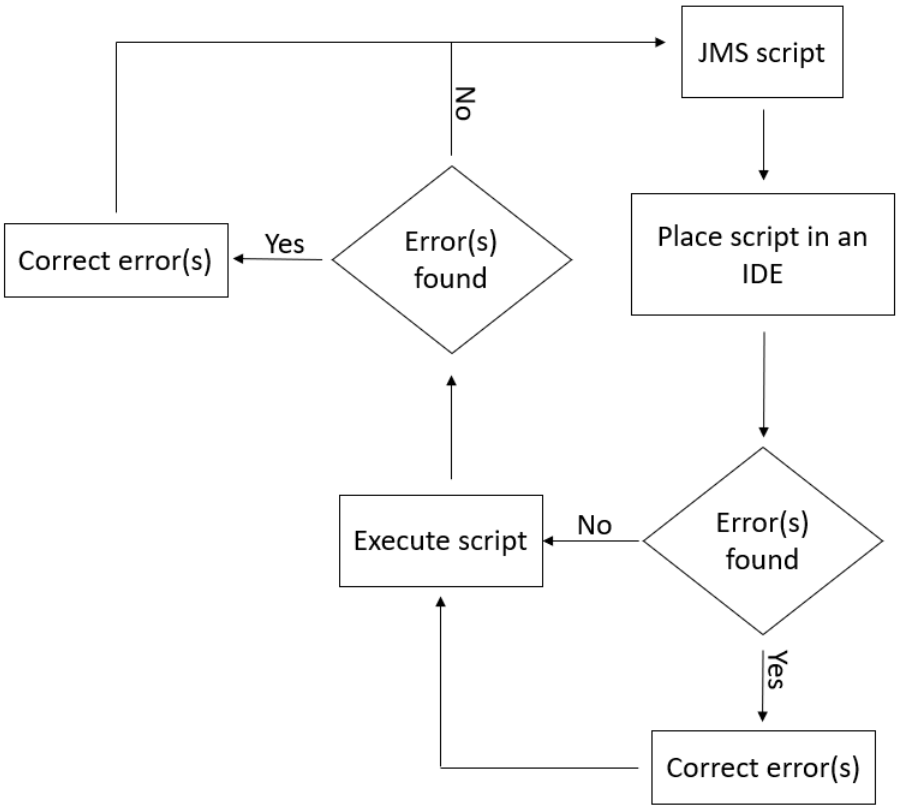

Figure 4.12: Flow chart for Python version 3.8 implementation

The process flow chart demonstrates how updating Python version 2.7 scripts to Python version 3.8 scripts was done using the combination of Integrated Development Environment tool and Python error detection systems .

Figure 4.12 shows how JMS Python scripts will be taken in and a process applied to those scripts to determine whether updates are required. Various required updates were identified using this process that needed to be addressed. Some examples of these needed changes, were syntax differences like how *os* library *chmod* function. For Python version 2.7, the *chmod* function does not use a lower case 'o' after the leading zero when setting file permission like in Python version 3.8 (figure 4.13 A). Therefore all the scripts in JMS which were setting permission using the *chmod* function had to be changed.

Additionally, print statements syntax also changed between Python version 2.7 and version 3.8. The compiler of Python version 2.7 accepted the print statement without brackets to encapsulate the output of the statement and usage of the $\rangle\rangle$ symbol to write print statement outputs into files. Lastly, the exception handling syntax is also different between the two Python versions. This can be seen in how the exception handling for the Python version 3.8 uses the *as* syntax to assign the error to the variable *ex*, while Python version 2.7 does not use

this syntax. Figure 4.13, shows examples of how the syntax differences previously mentioned were handled comparatively between the two versions of Python.

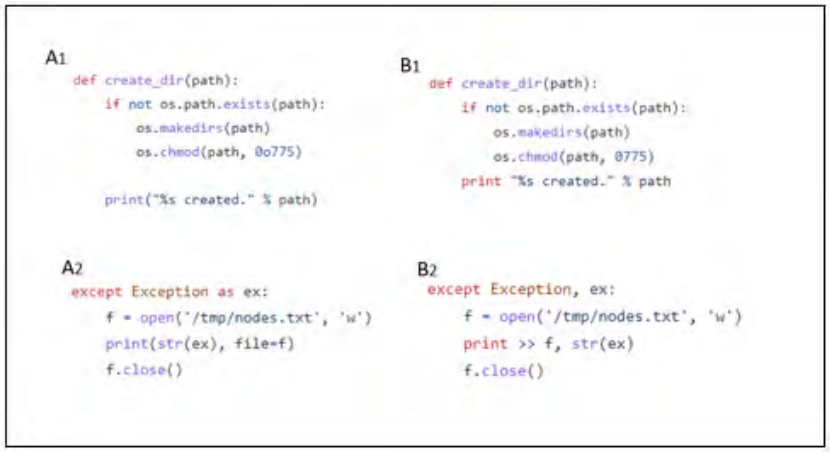

Figure 4.13: Examples of syntax differences between Python version 2.7 and 3.8

Python code illustrating the syntax dfferences between the two Python versions of JMS scripts.

A: Images represent Python version 3.8, implemented onto JMS.

B: Images represent Python version 2.7, implemented onto JMS.

1: Images show how the *os* library function *chmod* accept different permission syntax and print statement without the brackets encapsulation.

2: Images show difference in how exceptions are caught and also the difference in how the print statement is use to write print outputs into a file.

The secondary issue was the library retrieval and library deprecation differences between Python 2.7 and the selected Python 3.8. The first issue was the failure of Python 3.8 to retrieve the classes contained in local JMS scripts using the *import* library in the same fashion used by Python 2.7. The Python version 3.8, in this instance, does not directly import the scripts that contain the classes being imported. Various alternatives were attempted to address this issue, such as using the *sys* module, *path* class by adding these various directories containing scripts that have the classes being imported using the *amend* function.

However, making these directories available to the Python version 3.8 compiler in this instance did not address the issue. The solution was using the *from* Python functionality to specify the directories from which the scripts containing the classes are found. This changed how the local libraries (classes) are sourced and had to be done for all JMS scripts that import local libraries (classes). It is worth noting that just updating the syntactical differences between the two Python versions is tedious as each script that carries out that functionality has to be edited. Figure 4.14, shows the difference in the accessing of local scripts that contain classes between the two Python versions.

Figure 4.14: The difference in importing local class between Python 2.7 and 3.8

Examples of how the importing of scripts containing class being used in other JMS scripts differs between the Python versions 3.8 (**A**) and 2.7 (**B**) of JMS.

Another issue that had to be addressed was related to library deprecation and added library. It tends to be the case that with new releases of a given software tool, particularly programming languages, the new version will contain new functionalities and deprecate older functionalities. Various libraries had to be replaced from Python version 2.7 to 3.8. These libraries were identified using the same process shown in figure 4.12. For instances such as *os* module in Python version 3.8 using the *FileIO* function to implement the *RawIOBase* interface while for Python version 2.7 just using the *file* shorthand to carry out the that functionality.

In other instances, the library or class being imported from the module or library is no longer available in the updated version 3.8. An example of this instance is the encryption and decryption process used by the cryptography scripts of JMS. The *PKCS1_OAEP* class had to be imported from the *Cipher* library of the *Crypto* module for doing the process mentioned previously. Figure 4.15, shows the two examples of how the functionality of the scripts mentioned above was maintained using different resources.

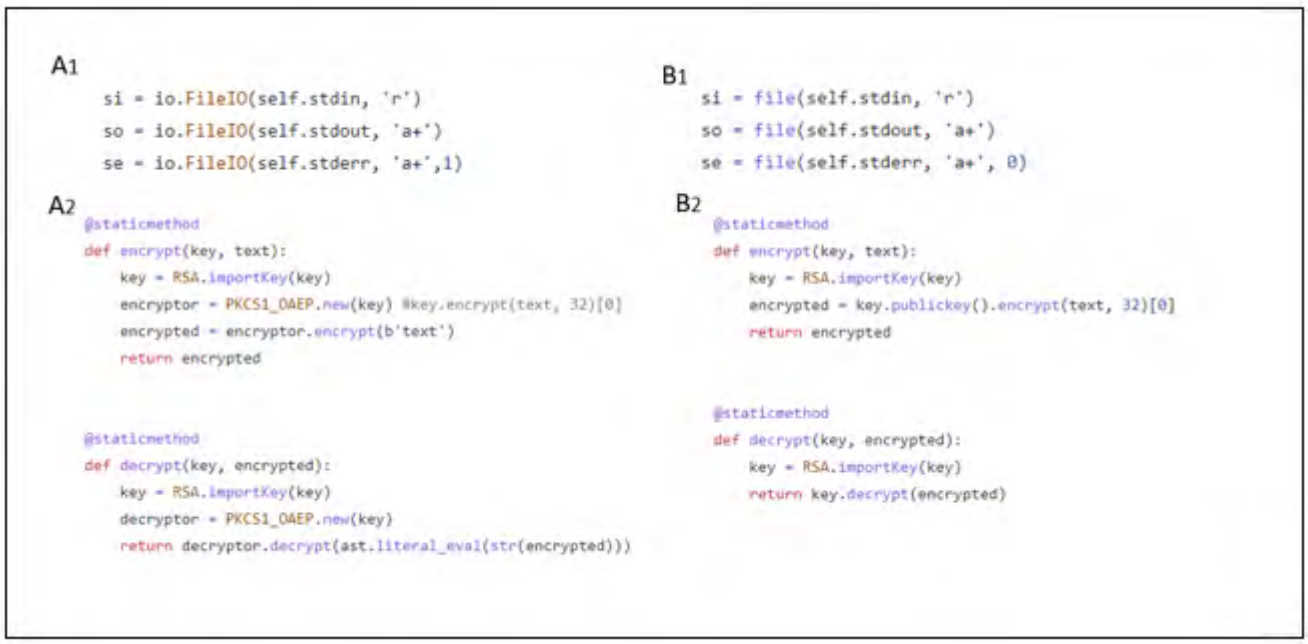

Figure 4.15: Different usage of Modules to accommodate Python version change to maintain similar JMS functionality

Examples of how different classes were used to maintain similar function due to Python version change.

A: Python version 3.8 code

B: Python version 2.7 code

1: The implementation differences in using the *RawIOBase* interface from the *io* module.

2: The usage of the *PKCS1_OAEP* class for cryptography to ensure no error is generated by Python 3.8 compiler.

Figure 4.15, shows the examples of the instances mentioned previously of the sourcing of new classes as a result of the ones being used not enabling similar functionality in the Python version 3.8 was being implemented. The issue surrounding the usage in terms of a different implementation syntax of classes contained in Modules (libraries) between the two versions, as addressed above, does not fully capture libraries' deprecation or implementation syntax. Figure 4.15, primarily shows deprecation of libraries in terms of their implementation syntax and briefly shows usage of different libraries, but both instances are within the same library as the *PKCS1_OAEP* class is from a library that already existed in Python version 2.7.

Therefore, to show how deprecation in regards to Modules (libraries) between Python version 2.7 and 3.8 was sourced. Python version 3.8 environment was created and named *venv* that will contain all the libraries that are used in JMS. The installation and creation process of this environment can be found in the Python documentation (Łukasz Langa, n.d.). Once this environment was created, all the libraries which were not deprecated were installed within this environment, followed by additional and replacement libraries for Python version 3.8.

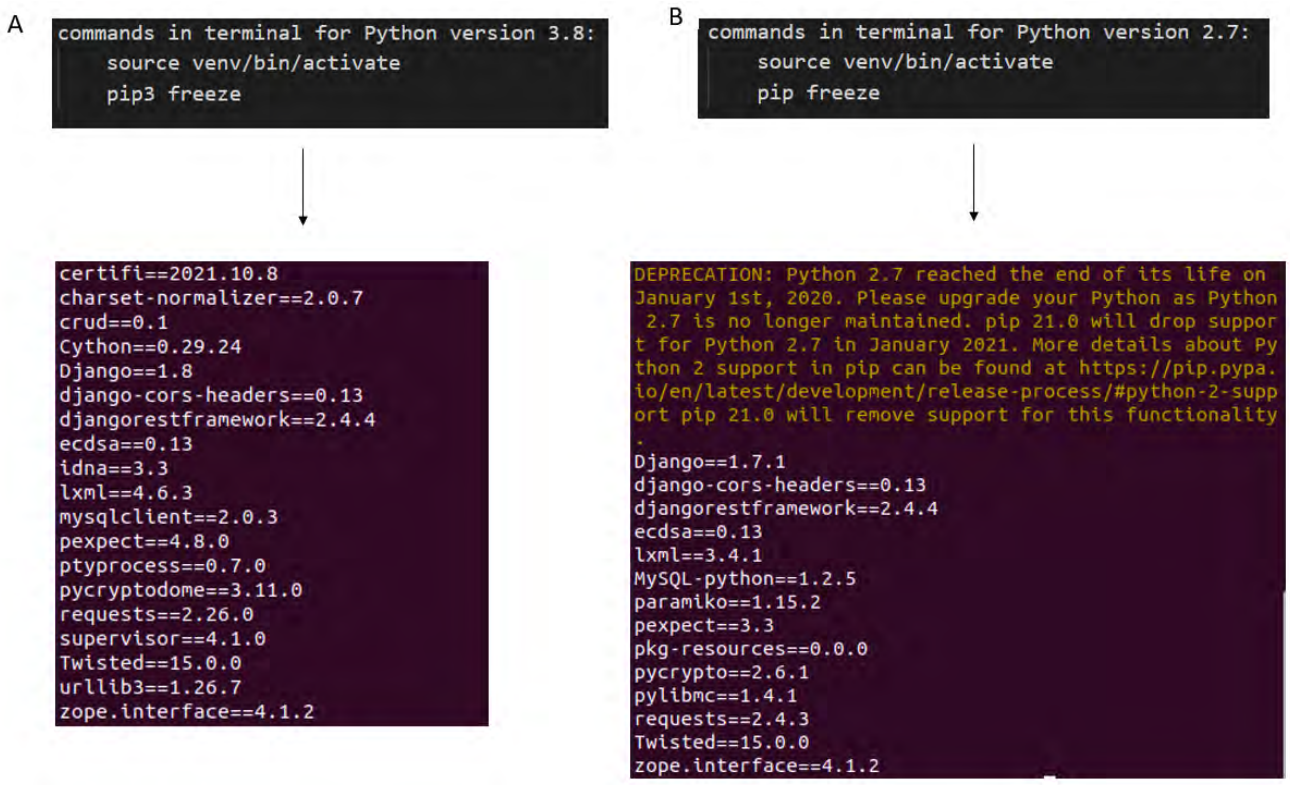

Figure 4.16: The libraries installed in the different Python virtual environments

The commands that are used to activated environments and display the libraries installed in the two Python environments for version 3.8 and 2.7. This process should be executed in the source(src) directory.

A: The process used to determine all the libraries installed in the Python version 3.8 virtual environment.

B: The process used to determine all the libraries installed in the Python version 2.7 virtual environment.

Figure 4.16, shows the library specifics for each of the Python versions in question. This shows that for libraries such as *lxml*, *pexpect*, *requests* and *Django* new versions of these libraries where sourced for the JMS Python version 3.8 (figure 4.16). In regards to deprecation, libraries such as *mysqlclient* was sourced to replace *MYSQL-python* and *pycryptodome* sourced to replace *pycrypto* (figure 4.16).

The remaining installed libraries are to support the functions carried out by existing libraries or are dependencies such as *certifi* which participates in processing web certificates. In contrast, a library like *charset-normalizer* is used in the encryption process of JMS (figure 4.16). *Crud* library was installed to assist *Django* default database functionality of using an SQLite database. The *Cython* library will make it possible for future end-users of JMS to write any low-level language such as C/C++ compiling scripts for JMS from the perspective of Python. All the functionalities of the given libraries can be found in the Python package repository, which will contain all the documentation of the set libraries seen in figure 4.16.

Figure 4.11 shows the environment where the existing JMS was cloned, and the above mentioned Python version 3.8 updates were done. It is worth noting that the JMS which was cloned from GitHub is different from the JMS currently in deployment in RUBi. The difference between the two JMS web applications was aligned during the Python version update. As previously stated, the environment was the Luke server that was configured to use the OpenPBS resource manager mentioned previously and the second JMS environment that uses Torque was required. That environment was the Chewbacca RUBi server which carries the JMS functionality using Torques. These environments were used comparatively to determine whether the changes in the resource manager on Torque JMS maintain similar functionality in OpenPBS JMS.

It is crucial to determine the back-end scripts that carry out the functionality of Torque. These scripts would probably contain the various Torque commands and provide insights into which commands from the OpenPBS resource manager can be used as possible replacements. Furthermore, these scripts would also process the Torque commands' various outputs for JMS consumption also need to be identified. These scripts would need to be changed in ways that process the output of the OpenPBS resource manager commands while retaining similar results used by JMS.

Other essential features of such scripts are relaying computer server information to the JMS web application, such as the jobs executing on a given compute server. To accommodate the new resource manager, all the scripts and relevant parts will need to change to ensure that JMS functionality is not fundamentally changed. Figure 4.17 shows a systematic identification process of these scripts mentioned above using a process chart.

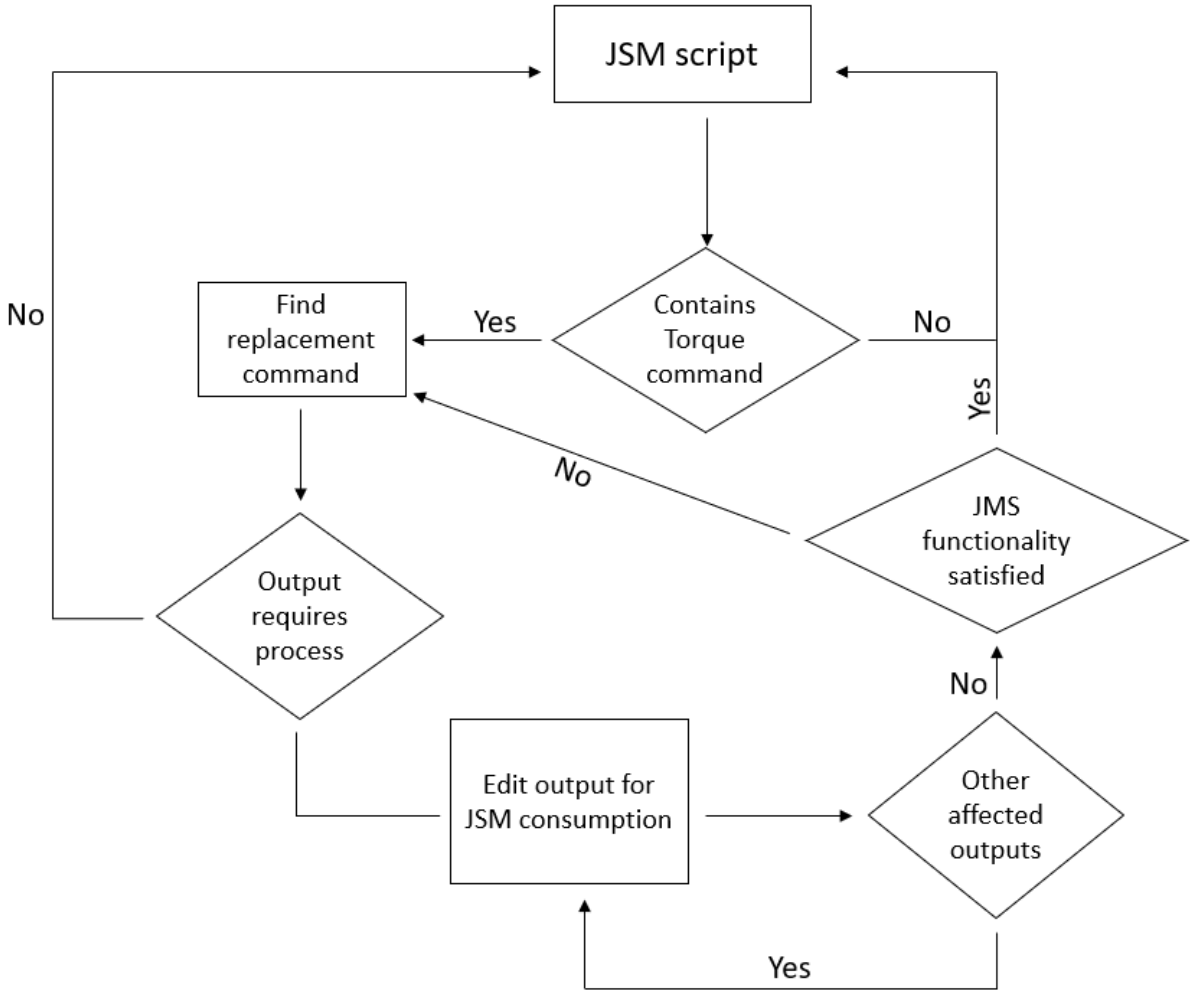

Figure 4.17: Flow chart for identification of JMS scripts that deploy Torque

Job Management System (JMS) is a web-based Workflow Management System for distributed computing servers architecture using Torque as its resource manager and job scheduling software. This process flow chart shows the steps to identify the parts of JMS that deploy Torque software in JMS for replacing those parts with OpenPBS resource manager and job scheduling software.

Once the scripts were identified that execute Torque commands, finding replacement OpenPBS commands can be undertaken. The directory containing scripts that execute Torque commands, including support functions, was the *src/jobs/JMS* from the root of the JMS root directory. Within this directory and its sub-directory, specific scripts execute Torque commands. Using the approach seen in figure 4.17, these scripts were identified as the *resource_managers/base.py* and *resource_managers/torque.py* that execute Torque commands. A process of identifying these replacement commands was designed to determine whether the output of the replacement commands OpenPBS is similar to the used Torque commands (figure 4.18).

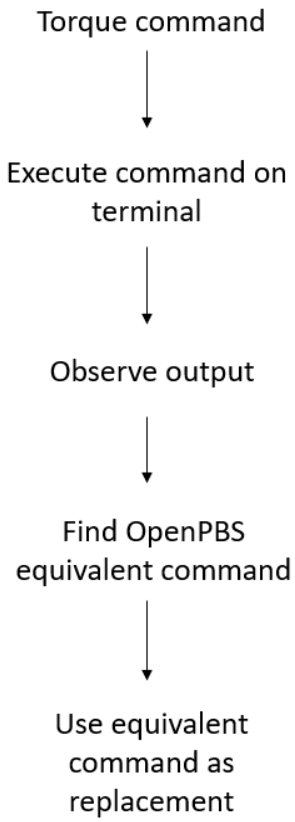

Figure 4.18: The process to identify OpenPBS replacement commands for JMS

This diagram shows how replacement OpenPBS commands were identified based on whether the OpenPBS command output has similar information as the Torque command in question. The comparison was made by executing the commands on the Luke and Chewbacca terminal to determine similarity based on the output content.

Figure 4.18, shows the process which was used to find OpenPBS commands meant to replace the Torque commands. However, to determine how the command's output will be processed in the JMS web interface, the JMS web application will need to be started. To start the JMS web application, a few requirements are needed. These requirements include setting the database used by JMS and ensuring that all the web requirements, such as Django web development framework packages, are set up.

A MySQL database was used as the database software for JMS. The MySQL software was sourced from the Ubuntu 20.04 LTS package repository and installed on the Luke server. Once the software was installed, the creation of the database, creating and populating the table within that database used by JMS are the few steps that need to be followed (MySQLTutorial contributors, 2022). Firstly, as per JMS requirements, a database named JMS was created, followed by creating a user named JMS that has access permissions to that database (MySQLTutorial

contributors, 2022). All of this was done in the terminal of the Luke server (figure 4.19).

```
############## MySQL installation #############

sudo apt install mysql-server

sudo mysql_secure_installation

sudo systemctl enable mysql

sudo mysql -u root -p

############## MySQL commands #############

mysql> CREATE DATABASE JMS;

mysql> CREATE USER 'jms'@'LukeIP' IDENTIFIED BY 'jms_password';

mysql> GRANT ALL PRIVILEGES ON JMS.* TO 'jms'@'LukeIP';
```

Figure 4.19: The commands used for the installation and setting up process of MySQL database

The commands which were executed on the Luke terminal to install a MySQL server that will host the database used for JMS and the MySQL commands used set the properties that will enable that usage by JMS.

Figure 4.19 shows how a MySQL server software was retrieved from the Ubuntu 20.04 LTS package manager, the security protocol was then executed to ensure the sovereignty of the database server, and then the MySQL server was enabled. Once the MySQL server was set up, the process of setting up the attributes required by JMS to access the database was undertaken as per JMS requirement (figure 4.19). The requirements are to create a database that will be used for the various data usage functions of JMS. Furthermore, a MySQL user was created that will be used to access the MySQL database and then granting all privileges on the created database to that user (figure 4.19). Once the database name and user were created, the Django *DATABASES* variable in the *setting.py* script was then updated to give JMS the database directives (figure 4.20).

Figure 4.20: Setting the database directives for JMS in Django setting Python script

The process of setting the database attributes that using the *DATABASES* variable. These attributes will enable JMS to interact with the database which its details are designed by the given attributes.

Figure 4.20, shows how JMS was connected locally to the installed MYSQL server. Once this connection was made, the process outlined in the repository of JMS was followed. This outline will create the tables and populate those tables JMS uses for its data storage and queries. The outline process also includes navigating to the source sub-directory in the JMS directory. These subdirectories will contain the *venv* directory, which contains all the configuration information for the Python version 3.8 virtual environment and also contains the *manage.py* script.

The *manage.py* script is the utility script used to carry out various Django functionality (Django Developers, 2022). Furthermore, the creation and population tables in the designated database it is also enabled by the *manage.py* script as stated in the JMS configuration outline on GitHub. However, to use this Django utility done by the *manage.py* script, the virtual environment first needs to be activated, and this will be followed by the instructions given in the JMS GitHub repository (figure 4.21).

```
cd /srv/JMS/src
source venv/bin/activate
python manage.py migrate
python manage.py setup
```

Figure 4.21: Terminal commands for setting up JMS database attributes

The commands executed on the terminal of the *Luke* server given in the JMS repository. These commands will give JMS the ability to create, populate and query tables it requires in the designated database.

Once all the above JMS requirements were met, the testing of the replacement OpenPBS commands can then be implemented. All the above mentioned are required to proceed with testing the selected OpenPBS commands. The testing of the JMS Django web application was done locally on the Luke server by hosting the JMS web application locally (Django Developers, 2022). Following the local hosting procedure of the Django framework, the *Luke* compute server IP address was used in a browser to access the JMS web application (Django Developers, 2022). The first point of interaction with the JMS web application is the login web page. Due to JMS web application being a cluster management system, web clients who have access to the JMS web application are also users who have access to the compute server (Brown et al., 2015). Therefore, the validation of web clients had to be changed in Python version 3.8.

The *subprocess* library enables terminal commands to be executed in Python scripts and the output of these commands to be stored in initialised Python variables. Due to the compute server also being the server that is hosting JMS in this instance. The Linux command *su [options] [username [arguments]]* was used, with the username and password given by the web client on the JMS login web page for authentication. This authentication process for web clients works hand in hand with the Linux user authentication process. Therefore, the JMS web client authentication Python script will attempt to login using the given web client credentials into the compute server. If successful, that implies that the web client also has access to the compute server, and access will be granted to the JMS web application.

Access into the JMS web application displays the Dashboard web page. This JMS web page has various useful indicators regarding the server system (Brown et al., 2015). Various additions had to be made to maintain the JMS functionalities mentioned above in OpenPBS using the replacement command and all the needed adjustments to accommodate the use of OpenPBS. The first part is to apply the replacement OpenPBS commands used to generate the information contained in the dashboard web page. For this task, two primary JMS Python scripts handle all the back-end functionality for the dashboard, namely *jobs/JMS/__init__.py* and *jobs/JMS/resource_managers/torque.py*. The *__init__.py* script uses the functions contained in the class defined in the *torque.py* script to carry out the JMS functions.

Although the selected replacement OpenPBS commands return similar output as the Torque commands in question. Before the replacement, OpenPBS commands could be applied. It was noted that the Torque commands are returned in Extensible Markup Language (XML) format. On the contrary, the OpenPBS commands in this instance are returned in byte format. Therefore, once the replacement commands are applied, the output will have to be edited. The edit is necessary because, as stated previously, there is a difference in the command output format. In the context of the JMS dashboard web page, which displays server information, figure 4.22 shows which OpenPBS commands were used to replace the Torque commands for functions that source the information contained in the dashboard.

Figure 4.22: Examples of a OpenPBS command used to replace a Torque command

The difference between how the *DataSection* data structure is populated using a Python dictionary that stores the output of the OpenPBS command used to replace a Torque command

A: The OpenPBS command used to get details about the queue, and the Python version 3.8 code used to process the command output.

B: The Torque command used to get details about the queue, and the Python version 2.7 code used to process the command output.

Figure 4.22 shows how the output of the Torque commands were returned in XML format by using the *-x* option directive. By ensuring that the output is returned in XML format, the Python *objectify* class from the *lxml* library can be used to format the XML output from the Torque command into a string (figure 4.22). However, in the case of OpenPBS commands, the technique mentioned above was not used. Therefore before the output of OpenPBS commands can be processed, it has to be decoded (figure 4.22). This is because the commands are returned in an encoded format (byte format). Therefore in order to process the command out, the output must first be converted into string format by using the *decode* function (figure 4.22). Once the command output is decoded into string format, further process of the output can take place.

As stated previously, the output processing in JMS ensures that the data from the various OpenPBS command that were implemented are packaged appropriately. The packaging processes enable the web pages of JMS that visualise server information and offer server functionality to get the desired information and functionality from the output of OpenPBS commands. An example of this is how the *_ParseJob* function was changed to process the replacement OpenPBS command *qstat -fxw job_id* which is used to store various information regarding aspects of the compute server and sent to the frontend. Figure 4.23, shows the example of the differences between how the data is stored from Torque JMS to OpenPBS JMS in the *_ParseJob* function that is called by the *GetJob* function.

Different data structures are used in the updated OpenPBS version of JMS. For instance, the usage of a Python dictionary data structure in *_ParseJob* function to store all the different attributes of the OpenPBS command executed in the *GetJob* function. The fields stored in this Python dictionary data structure are then used to populate the *DataSection* data structure which will be used in different web pages of JMS (figure 4.23). All the above show examples of how different challenges posed by updating the Python version and the job scheduler of JMS were handled. It is worth mentioning that the *echo "sleep 60"* command was used as the testing command to determine whether the JMS web application detects the submission of the jobs and relays all the Luke server information onto the web application. Furthermore, the add tool function of JMS was also tested.

Figure 4.23: Difference in how data from commands is stored in data storage structures of JMS between OpenPBS and Torque

This screenshot shows the difference of how the *DataSection* data structure is populated with command output between OpenPBS and Torque.

A: The *DataSection* data structure used in the backend updated OpenPBS version of JMS uses a dictionary fields to get populated.

B: The *DataSection* data structure used in the Torque version of JMS uses *deepgetattr* function to parse the converted XML attributes to populated *DataSection*.

However, these examples may not show the scale of the changes made to get JMS to perform its primary function of tracking jobs submitted on a given compute server and interacting with those jobs (Brown et al., 2015). Therefore, in order for the scale of the changes to be comprehended, the GitHub repository with the updated back-end JMS will need to be made available, so this comprehension can be possible. Furthermore, this will allow third-party testing to determine whether the updates stated in this section were done successfully. Lastly, periodically the *git push origin master* was executed on the Luke server.

4.2.2 Results

Allocation of the Luke server enabled the maintenance to be applied onto a system which can be used as a compute server that handles interactions with JMS for RUBi. Therefore the successful application of the maintenance objectives on the Luke server can be an indication of how the deployment of JMS would function in production. The creation of the Python version table enabled an overview perspective of all the Python versions with their attributes that were available for selection (table 4.1). The selection of Python version 3.8 was due to its depreciation date being set to be in 2024 (Łukasz Langa, n.d.). The depreciation date gives JMS back-end Python design a lifespan of over two years before a new version of Python should be implemented. However, this does not mean that no back-end development will be required for the next two years. Some maintenance may be necessary during that period, such as security requirements, added functionality, or general improvements. These security requirements can arise as vulnerabilities are found and reported in the bug logs.

Furthermore, Python version 3.8 was released in 2019, giving the developers and the community of users in this software a couple of months to identify any bugs or security issues that the early releases may have. Once again, this does not guarantee that no faults exist within Python version 3.8 but only states the interaction time the version has had to deal with any identified issues. Additionally, the amount of *Reported bugs logs* column shows that Python version 3.8 sits in the middle of the available Python versions (table 4.1). As stated previously, the reported bug logs tend to correlate with the release date of a given version and are typically not to the stability of a given version (table 4.1).

Having an experienced developer to assist in the maintenance process tends to enable the application of that maintenance aspect to be guided within a knowledgeable framework (Tian et al., 2021). This, as mentioned, is the reason for selecting OpenPBS as the replacement resource manager and job scheduler of Torque. Ubuntu 20.04 LTS Focal Fossa flavour OS was the latest version of the Ubuntu flavour when the maintenance process started. Therefore, it was sourced as the OS that replaced the Ubuntu 16.04 LTS Xenial Xerus. The replacement does not only serve as a new OS version that was used but also means that the Luke server will be provided with hardware and general maintenance till 2025 and, most importantly, receive security patches till 2030 as per the rollout plan of Ubuntu (Canonical Ltd, 2022). Due to the Luke server being exposed to the web and also being maintained for extensive use as a compute server, security patches are important to ensure all OS related vulnerabilities are patched (Gurunath, Agarwal, Nandi, & Samanta, 2018). However, these updates given by Ubuntu are

only effective if they are installed onto the Luke system. Therefore, for such system-level important updates to be implemented, a process of downtime planning or server switching should be concerned so the Luke server can incorporate these important updates into its system (Ahmed, 2019).

The updating of the Ubuntu 20.04 LTS package manager ensures that all the packages that are sourced from the package manager are all in their latest version (figure 4.10). Furthermore, the update also will check whether the package manager has all the packages available and if some packages are missing, the update will install them into the package manager repository (figure 4.10). The installation of the Ubuntu *build-essential* installs the libraries and resources required for the system to build Debian packages from various compilers and build scripts (figure 4.10). On the other hand, the installation of *soft-properties-common* enables simpler interaction with the repositories in the package manager that are from different providers (figure 4.10). The installation of the *git* software will give the Luke server system the ability to access a VCS such as GitHub for traceability of the maintenance process and further software developments (figure 4.10).

The testing of OpenPBS once it has been installed and using the given commands enabled the checking of whether the output of those commands are as expected, which would inform if the installation was successful. Whereby the *qstat -B* command indicated if the OpenPBS software has set up a queue. The execution of the *echo "sleep 60"* command followed by the *qstat -B* command would show whether the OpenPBS queue is being populated. The population of the OpenPBS queue indicated that OpenPBS is successfully keeping track of all the tasks(jobs) that are being submitted on the Luke server. The *pbsnodes -a* command indicated the Luke server OpenPBS configuration attributes, and these attributes aligned with the set attribute during OpenPBS installation, also showing that the OpenPBS installation was successful on the Luke server.

The installation of Python version 3.8 on the Luke server resulted in the Luke server system operating using Python version 3.8. The *python3-venv* package enables the creation of a Python virtual environment (container) which was used in the process of setting the virtual environment for JMS. Furthermore, the usage of a python container gives rise to the benefits of containers, whereby it was used as a configured environment that has all the libraries and resources used by JMS within a closed system (Koskinen, Mikkonen, & Abrahamsson, 2019). Lastly, the installation of *python3-pip* was used to retrieve the Python package manager for

version 3.*, giving the ability to retrieve all the packages that were used in the maintenance process (figure 4.11).

The installed *git* software enabled the creation of a local repository which was used for staging changes which have been applied to the JMS scripts, and these changes were pushed to the remote. This does not only enable the traceability of the maintenance processes but, as stated previously, gives a space where OpenPBS JMS can continue being maintained using a VCS (Tian et al., 2021). Furthermore, the JMS cloning gives a copy that can be used to apply the maintenance objectives.

The availability of the copy of JMS enabled the application of the identification process of the required Python version changes using the flow chart (figure 4.12). Visual Studio Code was used as the IDE and its error detection capability that underlined and highlighted all the syntax errors that were not compliant with the Python version 3.8 compiler (figure 4.24). On the other hand, if the IDE fails to detect an error, the Python compiler would not compile the given JMS scripts and give the appropriate error message (figure 4.24).

Figure 4.24 shows examples of the results of applying the methodology of using an IDE and the Python version 3.8 compiler to location sections of the JMS scripts that required updates. This resulted in the application of all the syntax, library, and class sourcing updates required for the JMS scripts to be compiled by the Python version 3.8 compiler. It is worth noting that the sourcing of the libraries and their respective functions was done using the Python version 3.8 documentation to determine what are the replacements for the version 3.8 (Python Software Foundation, 2018). This process resulted in the successful update of JMS scripts from Python version 2.7 to Python version 3.8.

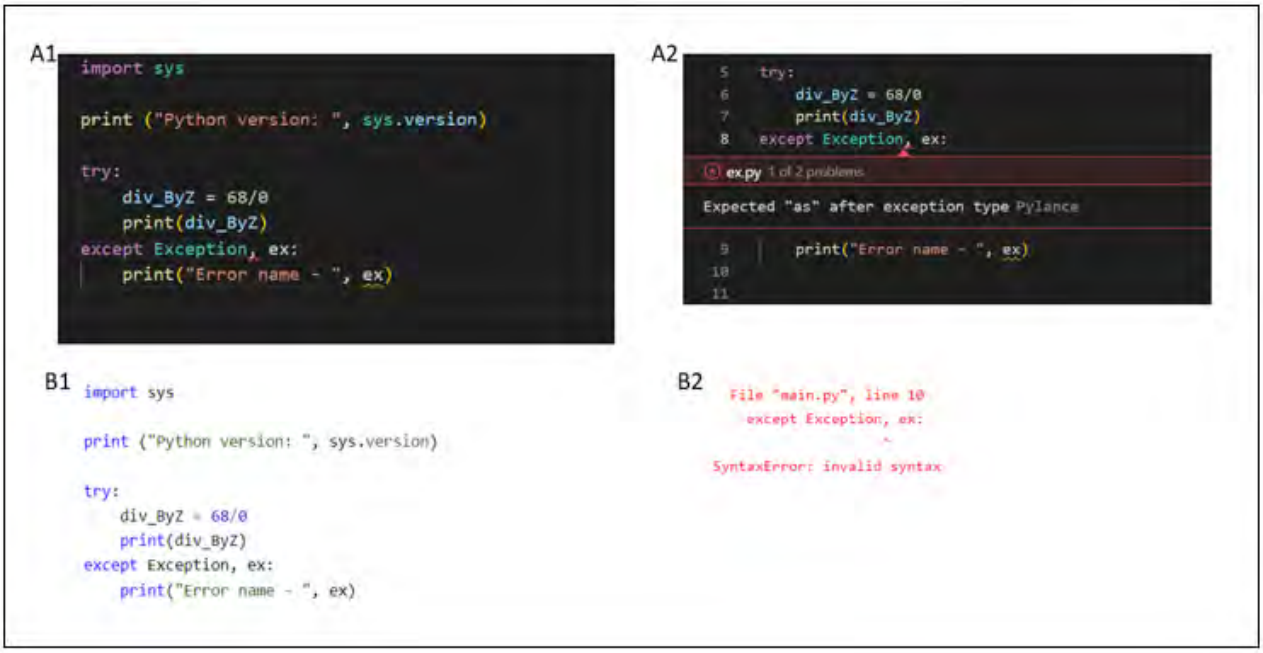

Figure 4.24: Examples of Python error detection using an IDE and the Python compiler

Examples of how Visual Studio Code Integrated Development Environment (IDE) and the Python version 3.8 compiler was used to detect errors in JMS scripts, to identify the changes that are required to update the JMS scripts to version 3.8 from version 2.7 of Python.

A: Visual Studio Compiler IDE error detection.

B: Python script and Python version 3.8 compiler error detection.

Additionally, given the big version gap between Python version 2.7 and the selected version 3.8 in this application context, the new features required for a version change get exaggerated. Once again, this highlights the need to ensure that a given entity's software is always maintained. This is to ensure that such exaggerations of having significant differences between version updates are minimised. Due to the content of a given version update being dictated by the owner of the software, not the end-user, having a periodic maintenance plan to apply the updates does not guarantee minimal changes. However, having a periodic maintenance plan to incorporate version updates is a good practice (Mohan & Greer, 2018; Talaei-Khoei, Ray, Parameshwaran, & Lewis, 2012). The updating of the JMS scripts to Python version 3.8 as the first part of the maintenance ensured that any further maintenance occurred on scripts that only required OpenPBS implementation.

Identifying JMS scripts that deploy Torque commands ensures that the type of Torque commands used are known, so their appropriate OpenPBS commands are found. The use of *Chewbacca* server to run the Torque commands provided the output information, which was then used to determine which OpenPBS commands contain the same output information. This resulted in the collection of OpenPBS commands that were used as replacements. The MySQL database and its attributes allowed for a local instance of the JMS web application to be hosted since having a database connected to JMS is required.

As previously mentioned, the local instance of JMS was used to test the select OpenPBS commands. Lastly, it is worth noting that the usage of a MySQL database during maintenance does not imply that MySQL use is mandatory for the JMS web application since any rational database software is sufficient for the data storage requirements of JMS. However, the Django web development framework must support that given database.

The application of the replacement OpenPBS commands, the added functionality to facilitate the use of those commands, and the testing thereof marked the completion of the objectives of the maintenance plan. JMS web application can now be classified as an open-source and free compute server monitoring and management web application. However, the management aspect is now limited.

Given that the OS which was used to carry out the maintenance plan will officially stop receiving security patches in 2030, the OpenPBS resource manager is open source and free, and lastly, Python version 3.8 has more than two years before its deprecation. All of this shows the lifespan of the JMS web application in deployment as a result of the maintenance that was undertaken. However, it is important to have an understanding that the status of all the tools being used can change, which would once again affect not only the classification of the JMS web application as a free open source software but also its usability. This highlights a point that has been continuously being stressed that various issues may arise from using software tools that are not in house (Stol & Ali Babar, 2010). Therefore a periodic status check of all the components being used is important to ensure there are no changes, and if there are changes, an appropriate maintenance plan can be formulated.

Lastly, the Cluster Management web page functionality of JMS was not updated to OpenPBS. The reason is that no alternative OpenPBS commands were found to replace the Torque command used on that particular web page. Due to the time scale of this thesis, no solution could be found. However, it is advised that a discussion should be had to revise this web page. This discussion should determine how the Cluster Management web page will be changed in JMS running in OpenPBS as an essential aspect of JMS functionality holistically. Secondly, security implementation comes into play, given the type of function this particular web page carries out. This is due to the Cluster Management web page giving the web clients using JMS the ability to make changes to the compute servers directly from the JMS web application. For some end-users of the JMS web application, this may be undesirable as a web client has the ability to set various configurations of the compute server.

Given that JMS also requires a front-end update of Django, as JMS is currently using Django version 1.8. whereby the Django web development framework is currently at version 4.xx. The old version of Django currently being used will not have all the security and vulnerability patches of the new version of Django. This may exaggerate the security implications of the Cluster Management web page, and since this section is only concerned with the back-end update of JMS, this is a task that will need to be addressed.

As stated above, the usage of the old version of Django does not prevent testing the updates done in this section. JMS is still being used in its old Torque and Django variation by RUBi. Therefore it can still carry out its function using the old version of Django once it has been configured and tested for application on RUBi compute servers. Furthermore, in the context of RUBi, JMS is not used to directly make changes to the compute server but rather as a tool used to observe the various attributes of the jobs submitted on the compute server(s). However, it is still advised that if the intention remains to publish JMS, a front-end update will still need to be done to ensure that even the front-end vulnerabilities are addressed before being released for public use as an open-source software. This front-end update can also be used to implement whatever discussion has been taken regarding the Cluster Management web page of JMS, as some users may want to use JMS as a tool to make changes to their compute server(s).

Chapter 5

Conclusion

For any interaction with web-related activities to be had, it was important to set out an understanding of the different aspects of the web. This understanding can be knowing the different categorisation of websites, their different classification and components. Such an understanding tends to inform how different web-related activities can be undertaken most productively (Smith, 2020). As briefly mentioned, hardware components provide the infrastructure so the web can exist, so understanding the hardware component is an important attribute to consider for a holistic approach to web development, and maintenance (Wang et al., 2004).

There was a lack of traceability regarding the websites and web applications hosted by RUBi. An objective was set out to develop techniques that can result in the traceability of RUBI websites and web applications. This was done by using written documentation and Virtual Control Systems (VCS) as techniques to achieve traceability. However, the documentation that was carried out was in regards to the storage server *Jabba* which is the server that holds the software tools and resources used by various RUBi web applications. In the context of web development and maintenance, it is important to document the server, which enables the web applications in question to function and as previously mentioned, this would then enable the replication of that server possible if a failure were to occur (Saadoon et al., 2021). Due to the storage limitation that was present during the documentation process, it is still important for this server to be backed up (Saadoon et al., 2021).

Furthermore, the *Jabba* storage server is relatively old, being over six years in service. Therefore, the hardware lifespan limitation should also be considered, given the age of the server and its important role. Although the documentation can allow for the storage server to be replicated, a backup server can enable any downtime caused by a failure on *Jabba* to be minimal (Saadoon et al., 2021). The minimal downtime effect of using a backup server that contains all the software tools and resources would be due to only a connect change being required. This connection change would be changing Padme Network File System (NFS) details to the be a client to the backup server, replacing Jabba while the failure is being addressed.

The created documentation can come in handy in this instance as it has all the NFS information required for the backed up data to be accessible to Padme. Therefore NFS information can be copied from the documentation sheet to the relevant NFS directory. However, for this to be done productively, the backup server would need the same directories as *Jabba* to ensure the directories stated in the NFS setup exist (Zakarya et al., 2012). Unlike an instance where the documentation is used to source all the software tools and resources stated, the backup server can be used. Where perhaps the documentation can be used to ensure that all the needed software tools and resources that may not be on the backup server can be sourced. In this context, it is advised that old servers such as *Jabba* are to be backed up and, in the best scenario, to be decommissioned so new servers can take up such important roles. Additionally, any new changes should also be added to the documentation to ensure it is updated and continues to produce value.

The use of VCSs for creating repositories that are used as storage and archiving of the developmental processes for all websites and web applications on the *Padme* served as part of the traceability objective. The backup system is very important as the *Padme* server, which hosts all the websites and web applications for RUBi is not backup. However, the repositories which have been created are only of value if they are used by continuously pushing updates into those repositories. The updating of the repositories does not only mean in terms of pushing software changes that may have been made. But as previously mentioned, RUBi web applications use various databases that web clients continuously update. Therefore, the failure to push periodic updates to the repositories will result in old repositories and defeating the traceability that has been achieved but also the backup capacity offered by the repositories (Spinellis, 2012).

Achieving traceability, particularly using VCSs like BitBucket and GitHub, ensured that any further development regarding the development and maintenance objectives were done using those platforms. The development of the COVIDRUG website using the Django web development framework enabled quick development of the website, as previously stated. This is due to the vast tools offered by Django and its usage in other RUBi websites that are in development, which offered a base of how best to achieve quick development of the COVIDRUG website. This was further assisted by the maintenance of the RUBi website, which contains similar attributes to the COVIDRUG website. Therefore, the maintenance process of the RUBi website aided the timely development of the COVIDRUG website.

The maintenance of MDM-Task and JMS demonstrated how maintenance processes could have different implementations and objectives. The maintenance of the MDM-Task was carried out to add new functionality and extend existing functionality. These additions and extensions enabled tools offered by the MDM-Task web application to be accepted as an academic work whereby its results are considered scientific (Amamuddy et al., 2021). Furthermore, the maintenance process was also used as an opportunity to carry out various optimisations and decoupling of the various components of the MDM-Task Web. The optimisation ensured that MDM-Task storage usage was minimised as it was mentioned that storage constraints were present during the implementation of the maintenance. Additionally, the decoupling techniques used ensured that future maintenance is simplified as some functionality is localised (Nnaa & Ojekudo, 2020).

This shows how maintenance implementation can also be used to address issues which were identified during the application of the initially set out maintenance plan. These optimisation and decoupling opportunities were identified during the maintenance and were incorporated into the maintenance plan. On the other hand, the maintenance that was done on JMS was changing software tools while maintaining the same functionality. This type of maintenance requires more understanding of the existing functionalities to ensure those functionalities are not changed. Therefore, more time was spent in the planning phase that was used to understand the different components that carry out the JMS web application functionality and the techniques used to identify these components.

The development of the identification techniques informed where changes needed to be made, what effect those changes would have and, most importantly, whether the given changes would result in the maintenance of JMS functionality while aiding the change of the software tools being used. This was important as implementation without appropriate planning could have resulted in a lengthened maintenance period due to a lack of understanding of JMS components and required changes (Tian et al., 2021). JMS was maintained with these identification techniques, and all the backend tools were updated and changed, ensuring that the set objective was achieved.

However, some aspects were not achieved, such as the *Cluster Management* web page. This was due to a lack of compatibility between Torque commands used previously in the *Cluster Management* web page that could not be directly translated into OpenPBS commands without fundamental changing functionality. For example, the *Cluster Management* web page has functionality like *TCP timeout (ms)* offered by Torque, but the same functionality is not available in OpenPBS. This incompatibility could not be addressed without changing the functionality of the *Cluster Management* web page, and these functions also have security implications. As stated, the prospects of the maintenance of JMS were to change backend software, not functionality. However, this does not hinder the function that JMS is carrying out in regard to its application in RUBi. Therefore, JMS can still serve its function in RUBi while the discussion around the *Cluster Management* web page is being had along with the frontend issues which were high lighted in the Results of JMS.

All these application processes were done, ensuring that the RUBi websites had all the relevant and up to date content regarding the research unit. At the same time, the COVIDRUG website provides information about the consortium and gives the consortium the potential to reach a wider audience with the benefits that come along with that reach. On the other hand, the web application maintenance shows how different maintenance can take place for web applications, resulting in MDM-Task receiving new functionality and attributes while JMS received a back-end software change and update along with all the additions required to accommodate those changes.

Lastly, the different prospects of this thesis showed that different approaches and solutions are available for dealing with web development and maintenance. From different tools to use in the implementation to different techniques that can be used in the approach. With all the moving parts, two priorities were identified, namely the need to keep track of tools that ensure sustainability, and the need to archive. The failure to use sustainable tools and archiving systems being used, particularly in the context of organisations and research units such RUBi can result in an unstable cluster system. This instability has implications for the satisfaction of the web clients who use the various deployed tools and the research capability of the unit (Al-Maskari & Sanderson, 2010).

References

Abriata, L. A. (2017). Web apps come of age for molecular sciences. In *Informatics* (Vol. 4, p. 28).

Ahmed, I. (2019). A brief review: security issues in cloud computing and their solutions. *Telkomnika*, *17*(6).

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., & Meedeniya, I. (2012). Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering*, *39*(5), 658–683.

Al-Maskari, A., & Sanderson, M. (2010). A review of factors influencing user satisfaction in information retrieval. *Journal of the American Society for Information Science and Technology*, *61*(5), 859–868.

Amamuddy, O. S., Glenister, M., Tshabalala, T., & Bishop, Ö. T. (2021). Mdm-task-web: Md-task and mode-task web server for analyzing protein dynamics. *Computational and Structural Biotechnology Journal*, *19*, 5059–5071.

Anaconda Developers. (2022). *Anaconda: Commercial Edition.* Retrieved from `https://docs.anaconda.com/anaconda-commercial`

Bauer, V., Heinemann, L., & Deissenboeck, F. (2012). A structured approach to assess third-party library usage. In *2012 28th ieee international conference on software maintenance (icsm)* (pp. 483–492).

Birnbaum, M. H. (2010). An overview of major techniques of web-based research.

Brown, D. K., Penkler, D. L., Musyoka, T. M., & Bishop, Ö. T. (2015). Jms: an open source workflow management system and web-based cluster front-end for high performance computing. *PLoS One*, *10*(8), e0134273.

Canonical Ltd. (2022). *The Ubuntu lifecycle and release cadence.* Retrieved from `https://ubuntu.com/about/release-cycle`

Casal, D. P. (2005). Advanced software development for web applications. *Looking ahead*, *15*(16), 16.

Chen, H., Yu, T., & Chen, J. Y. (2013). Semantic web meets integrative biology: a survey. *Briefings in bioinformatics*, *14*(1), 109–125.

Ciotti, M., Angeletti, S., Minieri, M., Giovannetti, M., Benvenuto, D., Pascarella, S., ... Ciccozzi, M. (2019). Covid-19 outbreak: an overview. *Chemotherapy*, *64*(5-6), 215–223.

Cock, P. J., Antao, T., Chang, J. T., Chapman, B. A., Cox, C. J., Dalke, A., ... others (2009). Biopython: freely available python tools for computational molecular biology and bioinformatics. *Bioinformatics*, *25*(11), 1422–1423.

Deepa, G., & Thilagam, P. S. (2016). Securing web applications from injection and logic vulnerabilities: Approaches and challenges. *Information and Software Technology*, *74*, 160–180.

del Pilar Salas-Zárate, M., Alor-Hernández, G., Valencia-García, R., Rodríguez-Mazahua, L., Rodríguez-González, A., & Cuadrado, J. L. L. (2015). Analyzing best practices on web development frameworks: The lift approach. *Science of Computer Programming*, *102*, 1–19.

Django Developers. (2022). *Django: Django documentation.* Retrieved from `https://docs.djangoproject.com`

Django Developers", t. . (n.d.).

Foster, I., Kesselman, C., & Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *The International Journal of High Performance Computing Applications*, *15*(3), 200–222.

Garett, R., Chiu, J., Zhang, L., & Young, S. D. (2016). A literature review: website design and user engagement. *Online journal of communication and media technologies*, *6*(3), 1.

Gnanam, S. P. (2016). Web engineering an overview and perspectives. *International Journal of Trend in Research and Development*, *3*(4).

Gurunath, R., Agarwal, M., Nandi, A., & Samanta, D. (2018). An overview: security issue in iot network. In *2018 2nd international conference on i-smac (iot in social, mobile, analytics and cloud)(i-smac) i-smac (iot in social, mobile, analytics and cloud)(i-smac), 2018 2nd international conference on* (pp. 104–107).

Ibrahim, I. M., Ameen, S. Y., Yasin, H. M., Omar, N., Kak, S. F., Rashid, Z. N., ... Ahmed, D. M. (2021). Web server performance improvement using dynamic load balancing techniques: A review. *Asian Journal of Research in Computer Science*, 47–62.

Jayamsakthi Shanmugam, D. M. (2008). Cross site scripting-latest developments and solutions: A survey. *Int. J. Open Problems Compt. Math*, *1*(2).

JS Knockout Developers. (2022). *Knockout JS documentation.* Retrieved from `https://knockoutjs.com/documentation/introduction.html`

Jurisic, M., & Kermek, D. (2014). Application framework development and design patterns: Current state and prospects. In *Central european conference on information and intelli-*

gent systems (p. 306).

Koskinen, M., Mikkonen, T., & Abrahamsson, P. (2019). Containers in software development: a systematic mapping study. In *International conference on product-focused software process improvement* (pp. 176–191).

Kubilius, J. (2014). i-review: Sharing code. *i-Perception*, *5*(1), 75–78.

Laato, S., Islam, A. N., Islam, M. N., & Whelan, E. (2020). What drives unverified information sharing and cyberchondria during the covid-19 pandemic? *European Journal of Information Systems*, *29*(3), 288–305.

Lawal, M., Sultan, A. B. M., & Shakiru, A. O. (2016). Systematic literature review on sql injection attack. *International Journal of Soft Computing*, *11*(1), 26–35.

Li, X., & Xue, Y. (2014). A survey on server-side approaches to securing web applications. *ACM Computing Surveys (CSUR)*, *46*(4), 1–29.

Li, Y.-F., Das, P. K., & Dowe, D. L. (2014). Two decades of web application testing—a survey of recent advances. *Information Systems*, *43*, 20–54.

Lynch, S. (2018). A tutorial introduction to python. *Dynamical Systems with Applications using Python*, 1–31.

MacFarlane, H., & Bultitude, M. (2012). The importance of websites. *Trends in Urology & Men's Health*, *3*(2), 33–36.

Meteor contributors. (2022). *Meteor Documentation.* Retrieved from `https://docs.meteor.com//full/`

Mohan, M., & Greer, D. (2018). A survey of search-based refactoring for software maintenance. *Journal of Software Engineering Research and Development*, *6*(1), 1–52.

MySQLTutorial contributors. (2022). *MySQL Tutorial for Ubuntu.* Retrieved from `https://www.mysqltutorial.org/install-mysql-ubuntu/`

Nguyen, V. N. (2017). A comparative performance evaluation of web servers.

Nnaa, S. B., & Ojekudo, N. A. (2020). A review of aspect oriented programming for enhanced program modularity. *International Journal of Computer Science and Mobile Computing*, *9*(1), 65-–74.

OpenJS Foundation and jQuery contributors. (2022). *JQuery Documentation.* Retrieved from `https://api.jquery.com/category/ajax/`

OpenPBS contributors. (2022). *OpenPBS Workload Manager.* Retrieved from `https://www.openpbs.org/`

Prokhorenko, V., Choo, K.-K. R., & Ashman, H. (2016). Web application protection techniques: A taxonomy. *Journal of Network and Computer Applications*, *60*, 95–112.

Python Software Foundation. (2018). *Python Documentation.* Retrieved from

https://docs.python.org/3/

Reniers, V., Van Landuyt, D., Rafique, A., & Joosen, W. (2019). Object to nosql database mappers (ondm): A systematic survey and comparison of frameworks. *Information Systems*, *85*, 1–20.

Rose, A. S., Bradley, A. R., Valasatava, Y., Duarte, J. M., Prlić, A., & Rose, P. W. (2018). Ngl viewer: web-based molecular graphics for large complexes. *Bioinformatics*, *34*(21), 3755–3758.

Ruby Rails contributors. (2022). *Rails Guides.* Retrieved from https://guides.rubyonrails.org/getting_started.html

Saadoon, M., Hamid, S. H. A., Sofian, H., Altarturi, H. H., Azizul, Z. H., & Nasuha, N. (2021). Fault tolerance in big data storage and processing systems: A review on challenges and solutions. *Ain Shams Engineering Journal*.

Salomon-Ferrer, R., Case, D. A., & Walker, R. C. (2013). An overview of the amber biomolecular simulation package. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, *3*(2), 198–210.

Sebastian, B. (2018). *PYTHON FOR BIOINFORMATICS*. New York: CRC Press.

Shetty, A. J., & Ganashree, K. (2020). Comprehensive review of datacenter architecture evolution. *International Research Journal of Engineering and Technology*, *7*(5).

SLURM contributors. (2021). *SLURM Workload Manager.* Retrieved from https://slurm.schedmd.com/

Smith, J. (2020). Teaching web development: A literature review. *EdMedia+ Innovate Learning*, 310–314.

Smits, P. A., & Denis, J.-L. (2014). How research funding agencies support science integration into policy and practice: an international overview. *Implementation Science*, *9*(1), 1–12.

Spinellis, D. (2012). Git. *IEEE software*, *29*(3), 100–101.

Stol, K.-J., & Ali Babar, M. (2010). Challenges in using open source software in product development: a review of the literature. In *Proceedings of the 3rd international workshop on emerging trends in free/libre/open source software research and development* (pp. 17–22).

Sun, K., & Ryu, S. (2017). Analysis of javascript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)*, *50*(4), 1–34.

Talaei-Khoei, A., Ray, P., Parameshwaran, N., & Lewis, L. (2012). A framework for awareness maintenance. *Journal of Network and Computer applications*, *35*(1), 199–210.

Tian, F., Wang, T., Liang, P., Wang, C., Khan, A. A., & Babar, M. A. (2021). The impact of traceability on software maintenance and evolution: A mapping study. *Journal of*

Software: Evolution and Process, *33*(10), e2374.

Venters, C. C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., ... Carrillo, C. (2018). Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software*, *138*, 174–188.

Wang, H., Huang, J. Z., Qu, Y., & Xie, J. (2004). Web services: problems and future directions. *Journal of Web Semantics*, *1*(3), 309–320.

Yousaf, N., Arshad, A., Nouman, M., & Arshad, U. (2018). Towards adaptive and responsive web design: A systematic literature review. *Language*, *1*, 40.

Yousaf, N., Butt, W. H., Azam, F., & Anwar, M. W. (2018). A systematic review of adaptive and responsive design approaches for world wide web. In *Future of information and communication conference* (pp. 704–717).

Zakarya, M., Rahman, I. U., & Ullah, I. (2012). An overview of file server group in distributed systems. *International Journal of Engineering and Technology*, *4*(6), 730.

Zhevaho, O. (2021). An overview of tools for collecting data on software development and debugging processes from integrated development environments. *Science and Transport Progress. Bulletin of Dnipropetrovsk National University of Railway Transport*(3 (93)), 24–37.

Zolkifli, N. N., Ngah, A., & Deraman, A. (2018). Version control system: A review. *Procedia Computer Science*, *135*, 408–415.

Łukasz Langa. (n.d.). *"python 3.8 release schedule.*