

School of Computer Science and Engineering Faculty of
Engineering
The University of New South Wales

Refactoring and Further Development of the Multilevel Genome Typing Database

Vandana Prathapar

Student ID: z5308943

Thesis Report submitted as a requirement for the degree of
Bachelor of Engineering in Bioinformatics Engineering

Submitted: 22nd November 2023

Supervisors: Prof. Ruiting Lan and Dr. Sandeep Kaur

Abstract

Bacterial pathogens have incredible genetic diversity, which has helped them infect humans readily throughout history. Multi-level Genome Typing (MGT) is a new technique that helps investigate the short and long-term epidemiology of bacterial species by assigning identifiers to the isolates based on the presence of genetic mutations in the isolate's genome. The Multi-level Genome Typing Database (MGTdb) is a web platform that was developed to store MGT sequence type (ST) assignments for a few bacterial species and provide interactive visualisations and summaries. Due to the limited number of MGTs available in the platform, the web code is also available in another codebase, to install on a local machine so that researchers can utilise the same visualisations for their own species of interest. However, the current codebase for MGTdb, is difficult to maintain and requires refactoring to help improve the process of future implementations for the platform.

This thesis describes the refactoring of the MGTdb codebase, as well as the modifications of the local installation process to account for the refactored code. The new codebase makes further developments and maintenance on the MGTdb more efficient. The refactored codebase was also moved to the local installation codebase to update the setup process and help simplify some of the steps involved. Simplifying the current local MGTdb installation process made a new user's experience less tedious and helped them understand the evolutionary relationships of isolates in their bacterial species of interest in a more seamless manner. Overall, this project helps MGTdb provide a better understanding of short and long-term epidemiology for many bacterial species, thus allowing them to find ways of minimising their spread.

Acknowledgements

First and foremost, I would like to thank my supervisors Prof. Ruiting Lan and Dr. Sandeep Kaur for their supervision throughout the year and inspiration for this project. The work I have accomplished this year would not be possible without you. I extend my thanks to the rest of the Lan Lab, for their feedback during practice presentations before my assessments. Thank you to my immediate family, parents, sister, and grandad for their endless support throughout this year. The homecooked food, banter and love have been pillars of strength for me. Thank you to the BABS honours group for the unforgettable lunches in the atrium; my BINF friends for exchanging rants when we needed to; my high school friends for the random catchups this year and my extended family for the events that forced me to take well-deserved breaks. I have also discovered that I have a taste for fancy cheeses and quince paste while doing this thesis so I would like to give a special shoutout to them. Finally, I would also like to thank Prof. Mark Tanaka for his assessment of my work throughout the year, which helped guide the direction of this project to where it is today.

Contents

Introduction.....	6
Background	8
2.1 Multi-locus Sequence Typing (MLST).....	8
2.2 Core Genome MLST.....	9
2.2.1 cgMLST vs MLST	9
2.3 Multi-level Genome Typing (MGT).....	10
2.4 Multi-level Genome Typing Database (MGTdb)	11
2.4.1 Data Retrieval	11
2.4.2 Data visualisations, Filtering and Downloads.....	12
2.4.3 Summary Reports.....	14
2.5.4 User Account Capabilities	15
2.5 MGTdb Database for an Organism.....	18
2.5.1 Isolate and Isolation Information	19
2.5.2 User Information	20
2.5.3 Extracted and Computed Information.....	20
2.5.4 Definitions and Schema	20
2.6 MGTdb Codebase	21
2.6.1 Architecture: Model, View, Template (MVT).....	21
2.6.2 MGTdb, The Codebase Currently.....	22
2.7 MGTdb read-the-docs Documentation.	23
2.8 Local Machine Installation of MGTdb	24
2.8.1 The Input files	25
2.8.2 MGT-local setup process	26
Challenges and Aims.....	28
3.1 Challenges.....	28
3.2 Aims.....	29
Refactoring the MGTdb Codebase.....	30
4.1 Methods.....	30

4.1.1 Setup of Testing Plan Before Code Changes	30
4.1.2 URL Re-routing	32
4.1.3 View Parameterising	33
4.1.4 Template and Static File Parameterising.....	35
4.1.5 Validation and Testing After Code Changes	37
4.2 Results.....	37
Simplifying and Updating the Local Installation of MGTdb	40
5.1 Methods.....	40
5.1.1 Simplify and update the setup and installation process.....	40
5.1.2 Update MGT-local and read-the-docs Documentation.	44
5.2 Results.....	44
Discussion	48
6.1 MGTdb Codebase Refactoring	48
6.1.1. Successful Refactoring of MGTdb codebase	48
6.1.2 Impacts of Refactored code for ongoing projects in MGTdb	49
6.2 Simplify and Update Setup and Installation Process and Documentation	50
6.3 Future Work	51
6.3.1 MGTdb Further Development	51
6.3.2 Further enhancements to Local Installation of MGTdb	52
6.3.3 Fixing Clonal Clusters for resolving Hierarchical Inconsistencies	52
Conclusion	54
References	55

Chapter 1

Introduction

Bacteria are diverse organisms that have pathogenic abilities to infect humans. Understanding the genetic diversity of bacterial organisms is imperative to understanding their epidemiology, whether long or short-term (Tankeshwar, 2021). A common technique for understanding the genetic diversity of bacterial species is sequence typing. Sequence typing is a technique that characterises isolates within a species using genetic information to generalise the distinction between different strains of bacterial pathogens (Maiden, et al., 1998). This helps confirm the evolutionary relationships between isolates in the species and describes the genetic diversity of the pathogen. In turn, this tells researchers how the pathogens evolve, giving insight into how to minimise the number of cases or infections caused by them (Maiden, et al., 1998).

One sequence typing method, Multi-locus Sequence Typing (MLST), was developed for typing *Neisseria meningitidis* in the late 90s (Maiden, et al., 1998). The technique aimed to resolve issues in the portability of previous characterisation techniques for bacterial isolates between other laboratories, due on an index variation (Maiden, et al., 1998). Since the invention of MLST, this provided an opportunity for researchers to develop on the MLST concept to create a variety of sequence typing methods that provided further insight into both long and short-term epidemiology.

Multi-level Genome Typing (MGT) is one of these methods (Payne, et al., 2020), that creates multiple MLST schemes of varying size to reflect short-term epidemiology, long-term epidemiology, and the epidemiology levels in between. The Multi-level Genome Typing Database (MGTdb) is a web platform for storing a variety of species' MGT schemes and the MGT assignment results that are produced for their isolates (Kaur, et al., 2022). The platform also provides interactive visualisations to understand the epidemiology of the species throughout the years and around the world, allowing users to generate summary reports and download information that's relevant to the species for further research.

The actual database for each organism is built on PostgreSQL and the codebase uses python with a django framework for the backend. The frontend is built with HTML, JavaScript, and CSS (Node.js, 2023). The structure of the codebase includes separate species folders that connect to their respective databases using the model, view, and template (MVT) software architecture. While each model is unique to map to the database information, the views and templates are like backend and frontend code where the files are continuously repeated throughout the platform. The software maintenance for the MGTdb codebase is very inefficient as a result.

This thesis project aimed to refactor the codebase for the MGTdb and improve the efficiency of future developments and bug fixes in the program. Rather than containing separate species folders in the platform with models, views and templates stored separately, the views and templates should be combined and pass the species name as a parameter in them, rather than contain multiple files of the same code functionality. The project also aimed to update the setup and local installation process of the local web application code to integrate the refactored code and simplify the process for a researcher using the web code.

This report is organised as follows; in **Section 2**, covers the background, which dives into how MLST was incorporated or developed on to make the MGTdb. In **Section 3**, the Challenges and Aims of this project are presented in more detail. In **Section 4**, I present how the first aim was achieved and in **Section 5**, I present how the second aim was achieved. **Section 6** is the overall discussion of this project and **Section 7** is the conclusions of the overall project.

Chapter 2

Background

2.1 Multi-locus Sequence Typing (MLST)

MLST is an *in-silico* method of identifying the evolutionary relationships between bacterial isolates (Maiden, et al., 1998). The first part of MLST, involves extraction of sequences with Polymerase Chain Reaction (PCR) Amplification with primers specific to the loci or genes that a researcher wants to use followed by Sanger sequencing (Larsen, et al., 2012). Once the sequences for the loci within all the isolates are obtained, one isolate is assigned as the ‘reference sequence’ with the allele numbers for each locus being assigned 1. From here, the other isolates’ sequences at each locus are assigned an allele number based on its similarity to the reference sequence and any previously compared sequences in that locus (**Figure 2.1**). The combination of the allele numbers forms an allelic profile, which is followed by assigning the sequence type (ST) of the isolate; a unique number that defines the isolate’s genetic composition in comparison to other isolates for this species. Isolates that have the same allelic profile will have the same ST value (Larsen, et al., 2012).

	Loci assignments				Allelic profile	ST
Reference	1	1	1	1	1-1-1-1	1
Isolate 1	1	2	1	1	1-2-1-1	2
Isolate 2	1	1	1	1	1-1-1-1	1
Isolate 3	1	2	2	2	1-2-2-2	3

Figure 2.1: Steps of MLST assignment for multiple isolates in a species. The above figure demonstrates the assignment of allele numbers based on the presence of a genetic variation (represented with different colours in the bars) in a locus position and how the number assignment is incremented based on previously compared isolates to the reference. The allelic profile and sequence type (ST) generation is shown on the right-hand side. Figure taken from Kaur, 2022 [Figure 2.1].

MLST was first applied to the *Neisseria meningitidis* species and was considered the “gold standard” of sequence typing (Larsen, et al., 2012). It was developed due to an inability to transfer the characterisation of pathogenic microorganisms to other laboratories, due to an index variation among other laboratories in strain classification.

MLST was considered an expensive technique back in early 2000s, as most of the cost comes from the extraction and sequencing processes for the pathogenic microorganism's data (Larsen, et al., 2012). However as sequencing cost came down in subsequent years, it gained wide popularity. One issue with traditional MLST is that it uses 7 genes from each isolate from an entire genome of the bacterial species. This captures only a fraction of the variation of the genome and thus only provides an understanding of the long-term epidemiology (Larsen, et al., 2012).

2.2 Core Genome MLST

With the rise of next generation sequencing and high throughput sequencing in the 2010s, whole genome MLST (wgMLST) and core genome MLST (cgMLST) rose to popularity. cgMLST involves taking a core set of loci, which is typically 90% of the genome, that reflect the genetic diversity of the species based on well characterised set of isolates (Ruppitsch, et al., 2015). This means there's a broader range of loci to compare within the isolates and the likelihood of observing mutations in the isolates increases. Using cgMLST allows the assignment of STs to isolates that have few allelic differences from one another. Because of the large number of loci or genes that cgMLST takes in, this helps identify closely related isolates, if the genetic code for two isolates is very similar and allows an observation of a mutation that occurred very recently (Achtman, et al., 2022).

2.2.1 cgMLST vs MLST

Table 2.2: Table comparing the standard MLST scheme with core genome MLST (cgMLST), based on factors such as: resolution, ST definition, Use, size of scheme and time of observing mutations for.

Property	MLST	cgMLST
No. of genes/loci	Uses 7 housekeeping genes/loci.	Uses a core set of genes/loci from the genome (1000+ genes)
Type of STs that are defined	Defines STs for historical clones	Defines STs for recent clones
Use	Long-term epidemiology.	Short-term epidemiology and outbreak detection.
Likelihood of observing a mutation in... (timeframes are dependent on the species)	100s years	8 months.
Resolution of result	Very Low	Very High

As seen in **Table 2.2**, mutations in bacteria isolates that occurred 100s years ago, or 8 months ago can be identified, but with these techniques alone, this does not provide the capability to view the relationships of the bacterial isolates at many resolutions. Therefore, there is need for a technique that covers a “spectrum between MLST and cgMLST” (Payne, et al., 2020).

2.3 Multi-level Genome Typing (MGT)

Multi-level Genome Typing (MGT) is a method for identifying genetic variation in bacterial isolates (Payne, et al., 2020). It is a combination of MLST schemes or MGT levels that contain an increasing number of loci to reflect the likelihood of observing a new mutation within a timeframe indicated by the level. MGT1 is the traditional MLST method applied on the isolate and the highest level of MGT for a species will contain the core genome MLST. This means that the lower levels of MGT produce lower resolution STs whereas the higher levels of MGT produce higher resolution results.

MGT was first developed for typing *Salmonella enterica* serovar Typhimurium (**Figure 2.3**). For this species, the MGT levels ranged from MGT1 to MGT9. MGT2 to MGT7 contained loci that were mutually exclusive to each other, to allow a deduction of the evolutionary relationships independently. The loci count increases between these schemes to reflect the increasing likelihood of observing mutations in the isolates. For example, MGT2 – 7 would reflect 100, 20, 10, 5, 2 and 1 year respectively for the average rate of an allele change (Payne, et al., 2020). MGT8 and MGT9 both use cgMLST at species and serovar levels respectively. MGT8 contained the core set of loci from the *Salmonella* species (Payne, et al., 2020) while MGT9 contained a larger number of loci, since some of them were specific to the Typhimurium serovar.

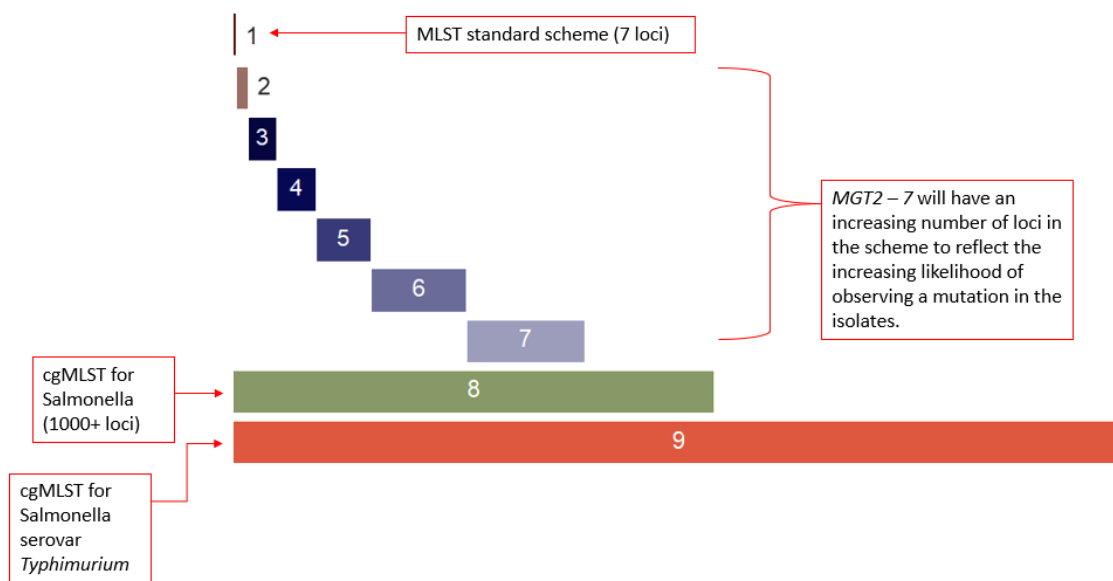


Figure 2.3: MGT for *Salmonella Typhimurium*. The diagram above depicts an increasing number of loci in each MGT scheme, to represent an increasing likelihood of observing a mutation. Figure taken from Payne et al, 2020.

The number of levels in a species' MGT depends on the diversity of the genome. *Salmonella enterica* serovars Typhimurium and Enteritidis both have diverse genomes and therefore would require 9 schemes whereas *Bordetella pertussis* MGT only requires 5 MLST schemes, because its genome is far less diverse. (Kaur, et al., 2022).

Another identifier produced by the results of MGT is the Genome Type (GT). GT is a combination of STs at the different MGT-levels. The length of the GT is dependent on the number of schemes that are present in that species' MGT. For example, since *Salmonella enterica* serovar Typhimurium has 9 levels, the GT contains 9 numbers separated by hyphens and in *Bordetella pertussis* there are 5 numbers. GTs provide a more precise definition of the isolate through the multiple resolutions of the genome (Payne, et al., 2020).

2.4 Multi-level Genome Typing Database (MGTdb)

The Multi-level Genome Typing Database (MGTdb) is a web platform that stores the STs for isolates across a set of MGT schemes of a species (Kaur, et al., 2022). The platform also enables uploading genome data as sequence reads for alleles that can be processed and have MGT identifiers assigned to them. MGTdb also provides visualisation tools that summarise information based on parameters in a digestible way to a researcher to interpret their results (**Figure 2.4**). Currently, there are MGTs for *Salmonella enterica* serovar Typhimurium, *Salmonella enterica* serovar Enteritidis, *Vibrio cholerae*, *Bordetella pertussis* and *Staphylococcus aureus*.

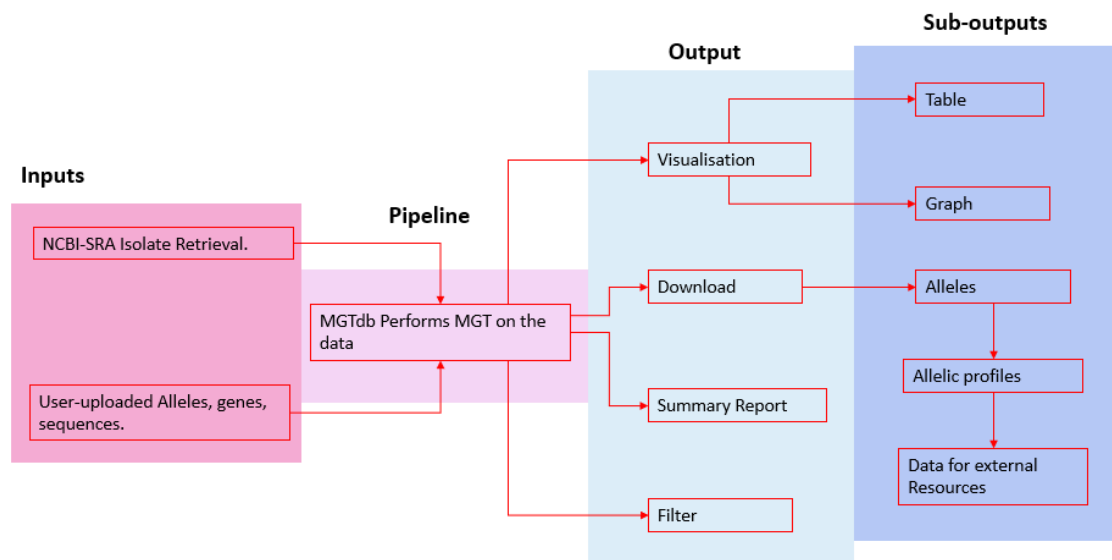


Figure 2.4: A summarised workflow of the MGTdb platform. The pink section is data retrieval and sequence read; the purple section shows the MGT assignments being made. The light blue section shows the functional properties of MGTdb, after the MGT assignments are made and the dark blue section shows the outputs from these functions.

2.4.1 Data Retrieval

There are two main sources of isolate data which is processed by MGT. The first one is by the user uploading their data as sequence reads in fastq.gz files to be processed by the platform and have its MGT assigned accordingly. This process requires authentication and an account with MGTdb. The second source is the National Centre for Biotechnology Information – Sequence Retrieval Archive (NCBI-SRA) which allows the creation of a central database out of the isolates (Leinonen, et al., 2011).

The NCBI-SRA is a database containing many DNA reads that are set up for the most species that have available MGTs. The reads from this database are set up to retrieve data every 24 hours and pass the reads into the MGT workflow to assign MGT-STs (Kaur, et al., 2022).

2.4.2 Data visualisations, Filtering and Downloads

Once the MGT STs are assigned to the uploaded isolates, they are displayed in a table with all the isolates (**Figure 2.5**). The table cells are also coloured based on the value of the MGT ST for easier distinction between the STs throughout the species' isolates. The table also contains additional metadata related to the source of the isolate and year it was collected in (**Figures 2.6, 2.7, 2.8**). This table's entries can also be sorted based on its columns by clicking the title of the button. This results table can also be filtered in the top of the page by the different columns in the table.

The table containing all isolate results for a species' MGT also has the capability of viewing the isolates based on the clonal complexes, which are values that are subject to change based on new isolates being retrieved from NCBI-SRA or user uploads. The isolate table can also display a 'complete' ST, which is a decimal value of the normal ST that reflects whether there is any missing data in an isolate's genome that might infer a chance that the isolate shouldn't be assigned the ST it currently has.

We can also look at a species' STs through a graphical view. The interactive graphical visualisation summarises the counts of STs, CCs and ODCs at all MGT levels simultaneously without additional metadata, using only MGT assigned isolates (Kaur, et al., 2022). Graphical visualisations of 'time and location' and 'time or location' are also present in the platform.

Results for a species MGT can also be downloaded as an entire .csv file or with the filtered results. Some additional publicly available data to the users includes the allelic profiles of the largest scheme in the species' MGT and alleles can be found on the website.

List of isolates

Sequence types view Clonal complexes view Graphical view ⓘ

Display complete ST Display color ⓘ View complete ST and colour Isolate table

Total isolates found: 61332
Page 1 of 614 next (2) last (614) Enter page number: Go ⓘ

Isolate	Server status	Assignment status	project	Privacy status	Run as query	Serovar	MGT1 ST	ST- MGT2	ST- MGT3	.. More columns
SRR5642043	Complete	Assigned MGT		Public			513	155	725	
SRR3622956	Complete	Assigned MGT		Public			19	1	1	

More isolates

Figure 2.5: Isolate table in the platform that displays all the MGT ST assignments for all isolates in *Salmonella enterica serovar Typhimurium*. This page also allows users to select a ‘clonal complex’ view or ‘graphical view’ of the STs, CCs, and ODCs for the isolates that have been uploaded for this species and the table provides extra metadata such as the source and year of collection.

Isolate	Server status	Assignment status	project	Privacy status	Run as query	Serovar	MGT1 ST	ST-MGT2	ST-MGT3
SRR11233322	Complete	Assigned MGT		Public			19	32	2
SRR5132034	Running reads to alleles	-		Public				-	-
SRR1198924	Complete	Assigned MGT		Public			19	5	756
SRR5090732	Complete	Assigned MGT		Public			19	1	1
SRR1177215	Complete	Assigned MGT		Public			19	2	11
SRR4427819	Complete	Assigned MGT		Public			99	58	564

Figure 2.6: First section of the isolate table that contains: Isolate identifier, Server status, Assignment status, project, privacy status, Run as query, Serovar and MGT1 – MGT3.

ST-MGT4	ST-MGT5	ST-MGT6	ST-MGT7	ST-MGT8	ST-MGT9	ODC1	ODC2	ODC5	ODC10	Continent	Country
203	17703	29446	39347	52851	57927	55831	46878	38454	29743	North America	Canada
-	-	-	-	-	-	-	-	-	-	South America	Argentina
1556	2422	3752	4613	5411	5719	5719	5719	5719	5719	North America	USA
3375	5352	8363	10750	13341	14144	13625	13288	12585	11587	Oceania	Australia
1539	21	3710	4557	5350	5657	5657	5657	5657	5657	North America	USA
740	1539	2359	2902	3422	3652	3652	3652	3652	3652	South America	Argentina

Figure 2.7: Second section of the isolate table that contains: MGT4 – MGT9, ODC1, ODC2, ODC5, ODC10, Continent and Country.

<u>State or sub country</u>	<u>Postcode</u>	<u>Source</u>	<u>Type</u>	<u>Host</u>	<u>Host disease</u>	<u>Collection date^v</u>	<u>Collection year</u>	<u>Collection month</u>
		feces		Bovine		2007-07-03	2007	7
Capital Federal						2007-07-03	2007	7
MN		puffed vegetable snack				2007-07-05	2007	7
		Faecal Sample				2007-07-06	2007	7
MN		Bos taurus				2007-07-06	2007	7
Buenos Aires		beef				2007-07-11	2007	7

Figure 2.8: Third section of the isolate table that contains: State or sub country, postcode, source, type, host, host disease, collection date, collection year and collection month. Some metadata information for the isolates are missing.

2.4.3 Summary Reports

Any user of MGTdb, can generate a summary report for the species they're investigating, by the country, project (if authenticated) and the years they want to investigate (**Figure 2.9**). This summary report shows the trends of the most common STs observed in static visualisations. Additionally, the latest data is provided each time a summary report is generated with all the available public isolates to give a snapshot of the trends at a particular time. This report can also be saved as a html document and downloaded.

Download a summary report ⓘ

Instructions

1. Choose a country. If logged in, can alternatively choose a project.
2. Choose a starting and an ending year.
3. Download the report (scroll all the way to the end to find the download button).

Country	Argentina ▼
Project	---- ▼

Year start	Year end
2014 ▼	2023 ▼

Note: if both project and country are selected, results with the conjunction of the two will be returned.

Generate report

Figure 2.9: Section in MGTdb, that allows a user to generate a summary report. In the above example, the user is logged into the platform, which gives them the ability to generate a summary report based on a project as well, but if the user is unregistered, this option would not appear.

2.4.4 User Account Capabilities

Users can hold an account with MGTdb, which gives them the capability of uploading their own isolates for having MGT STs assigned to them. Once an account is created, they will have access to the ‘Projects’ tab on the platform which gives them the number of projects they have under a species and the isolates in those species as well (Kaur, et al., 2022).

By creating a project, the user should be able to upload their own isolates within that project (**Figure 2.10 and 2.11**). They can either upload an individual isolate or multiple isolates. In uploading individual isolates, the necessary files include a ‘file forward’ and ‘file reverse’ file, which gives the sequence reads that we’re performing MGT on. Additionally, the user will have the choice for their isolate to be public or private, which will disable any of the MGT metadata or values from being shown to the public. This means this uploaded isolate can only be searched for by the authorised user. Uploading multiple isolates requires an .csv file containing the metadata for the isolates that are being uploaded and the isolate themselves.

Once the isolates are uploaded, the user is taken to the ‘isolate detail’ page, which shows the fields that were entered for the isolate. These fields are clickable and perform a search of the larger database, for isolates that have those fields in common with the uploaded isolate.

Create a new isolate








Isolate <i>i</i> *	<input type="text"/>
Privacy status <i>i</i> *	<input type="text" value="-----"/>
File forward <i>i</i>	<input type="button" value="Choose File"/> No file chosen
File reverse <i>i</i>	<input type="button" value="Choose File"/> No file chosen
Alleles file <i>i</i>	<input type="button" value="Choose File"/> No file chosen
Project <i>i</i> *	<input type="text" value="testproject"/>
Run as query <i>i</i>	<input type="checkbox"/>

Location details

Continent <i>i</i> *	<input type="text" value="-----"/>
Country <i>i</i> *	<input type="text" value="-----"/>
State or sub country <i>i</i>	<input type="text"/>
Postcode <i>i</i>	<input type="text"/>

Figure 2.10: First part of the form to upload isolates into a project. The main requirement for an isolate upload includes: the ‘File Forward’ and ‘File Reverse’ sections which contain files in the format of Isolate_X.fastq.gz. Mandatory fields for an isolate upload are marked with an asterisk (*) next to them.

Isolation details

Source 	<input type="text"/>
Type 	<input type="text"/>
Host 	<input type="text"/>
Host disease 	<input type="text"/>
Collection date 	<input type="text"/>
Collection year  *	<input type="text" value="-----"/>
Collection month 	<input type="text" value="-----"/>

Save

Figure 2.11: Second part of the form to upload isolates into a project. Mandatory fields for an isolate upload are marked with an asterisk (*) next to them.

2.5 MGTdb Database for an Organism

The database for MGTdb can be split into 4 categories: Isolate and Isolation information, User information, Extracted and Computed information and Definitions and Schema information (**Figure 2.12**). Most of the entities in this database contain information that is accessible through the platform, the rest of them are used in MGT assignments for uploaded or retrieved isolates.

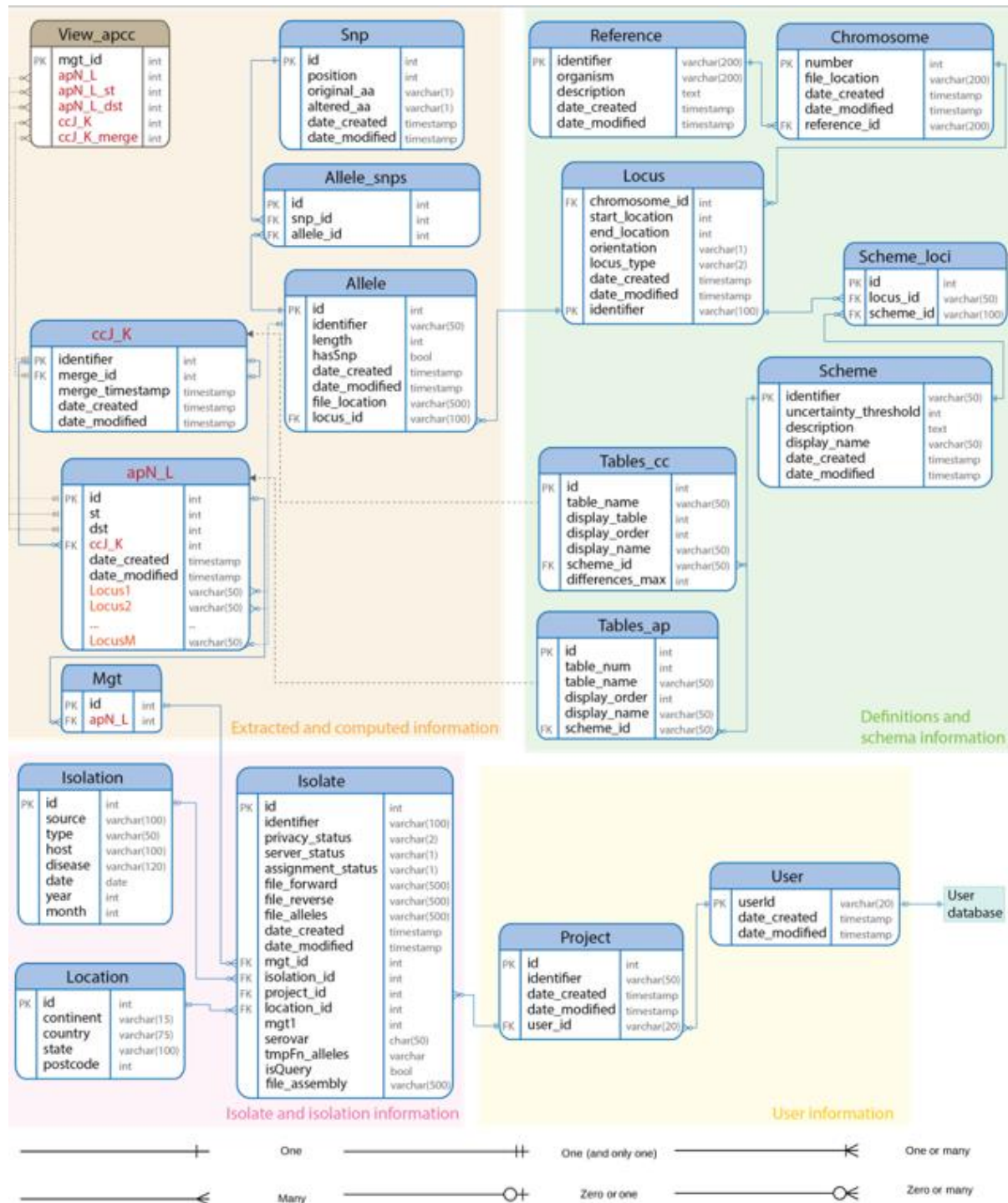


Figure 2.12: Entity Relationship diagram of the MGTdb database for a specific organism. The boxes are the entities, and the arrows represent the relationships these entities have. Figure taken from Kaur, 2022.

2.5.1 Isolate and Isolation Information

The isolate and isolation information of the database stores metadata pertaining to the isolates. There are three entities in this section: Isolation, Isolate and Location. The location has a one-to-many relationship with the isolate table since many isolates can be from the same location. Similarly, since the isolation table provides information about the timeframe and source of an isolate, this also has a one-to-many relationship with the isolate table. The isolate table stores information its status in the MGTdb pipeline and the further details that would be displayed in the isolate-detail page (**Figure 2.13**). Each of the entities also contain an 'id' or primary key, which is a unique identifier of the entry in the table, which helps routing for objects in the table.

Isolate details: SRR1555304 ⓘ

Isolate	SRR1555304
Server status	Complete
Assignment status	Assigned MGT
project	-
Privacy status	Public
Run as query	-
Serovar	-
MGT1 ST	19
Continent	-
Country	missing
State or sub country	-
Postcode	-
Source	-
Type	-
Host	-
Host disease	-
Collection date	-
Collection year	-
Collection month	-
Sequence types	ST-MGT2 ST-MGT3 ST-MGT4 ST-MGT5 ST-MGT6 ST-MGT7 ST-MGT8 ST-MGT9
	1 1 1 1 1 1 1 17019.1 <input type="checkbox"/> (Only include complete STs) ⓘ
Clonal clusters	CC-MGT2 CC-MGT3 CC-MGT4 CC-MGT5 CC-MGT6 CC-MGT7 CC-MGT8 CC-MGT9
	1 1 1 1 1 1 1 16247
Outbreak detection clusters	ODC1 ODC2 ODC5 ODC10
	16247 15692 14224 12427

Search Clear selection

Figure 2.13: Isolate details on the platform which are stored in the Isolate and Isolation Section of the MGT database.

2.5.2 User Information

The user information section of the database is used for storing new users that sign up to the platform and the projects they create. The two entities in this section include: User and Project. User to Project have a one-to-many relationship and project to Isolate has a one-to-many relationship as well. The user entity further connects into the user database which contains information that the user inputted about their account such as emails and passwords.

2.5.3 Extracted and Computed Information

The Extracted and Computed Information section of the database contains 7 entities: View_apcc, Snp, Allele_snp, Allele, apN_L and Mgt. The data that is stored in these tables are displayed throughout the pages of the platform and are carried by ‘models’ to be rendered into the frontend of the platform.

The Snp, Allele_snp and Allele tables all contain the gene sequences that are commonly used in the isolates for comparing any uploaded isolates to, before assigning any MGT. The apN_L table stores the allelic profile for each isolate at the different MGT levels that exist for the species. The number of loci columns vary, depending on the number of loci taken in each MGT scheme. In apN_L, N represents the MGT scheme that the table is for, and L represents the table number for that scheme. Some schemes have multiple ap tables since there’s a limit of 1000 columns for each table, and the higher level MGT schemes can take up to 5000 loci. The ccJ_K table stores the clonal complex values for the isolates where J is the MGT level and K is the table number. The Mgt table at the end of the program simply takes in the apN_L values to be connected to the isolates themselves.

2.5.4 Definitions and Schema

The Definitions and Schema Information of the database is used in the MGT scripts to assign MGT to the retrieved or uploaded isolates. The entities in this section include Reference, Chromosome, Locus, Scheme, Scheme_loci, Tables_cc and Tables_ap.

The Reference table simply stores the reference isolate for the species. It has a one-to-many relationship with the Chromosome table, which is used to store different chromosomes. The need for this table arose when certain species such as *Vibrio cholerae* were found to have two chromosomes instead of one. This chromosome table has a one-to-many relationship with the Locus table which stores all the possible loci that are used for assigning MGTs in this species, including the start and end positions of the locus.

The scheme table stores the possible levels of MGT for a species, including the display name and the uncertainty threshold. There is also a Scheme_loci table in the database to connect the scheme tables and loci tables easily. This is because the Locus and Scheme tables have a many-to-many relationship, because a locus can belong to multiple schemes and a scheme will contain many loci. Trying to join these tables will result in a lot of duplicate data which makes performing MGT a lot more complicated here. To resolve this, the Scheme_loci table was created to contain a primary key as well as the locus id and scheme id.

The Tables_ap and Tables_cc store the table numbers and names for the apN_L and ccJ_K tables, since there can be multiple allelic profile tables for an MGT scheme, due to the limited column capacity.

2.6 MGTdb Codebase

The database is built using PostgreSQL and the backend is coded in Python 3.8 along with a Django framework (Django, 2023). The frontend is built using JavaScript (JS), CSS and HTML (Node.js, 2023).

2.6.1 Architecture: Model, View, Template (MVT)

The Model, View, Template (MVT) architecture, is a software design pattern that enables efficient communication between a web platform and database (Plekhanova, 2009). The communication between these entities for web applications is demonstrated in **(Figure 2.14)**. This design pattern is built of the Django framework which implements a way of allowing the backend to access the database and pass the retrieved information into the frontend.

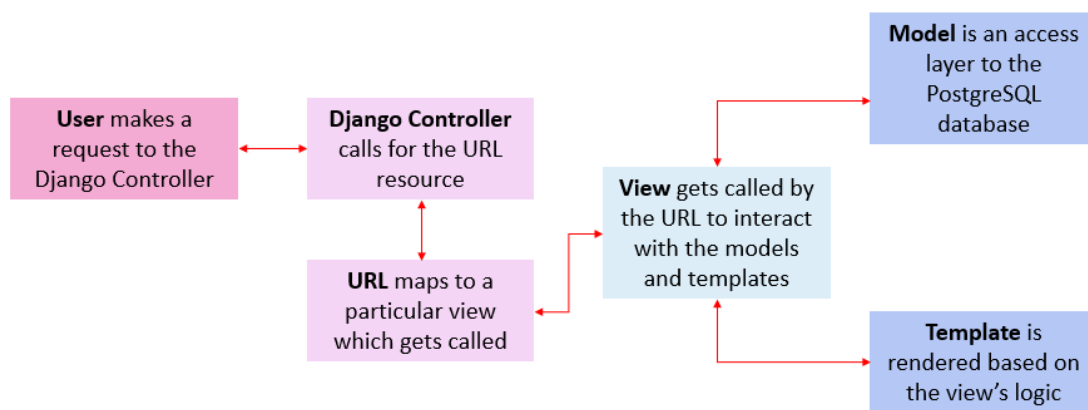


Figure 2.14: Workflow of the Model, View, Template (MVT) software architecture. Requests from a user are passed into a Django Controller which then calls a URL that is mapped to a particular view. The view then carries information from the models and executes business logic in the backend to render the templates.

The models are a data access layer between the database and the backend (Django, 2023). Models are python classes where each python class represents a table in the database and each attribute inside the class is a column from that table. For example, for the location table in an organism's database, a python class would be created called 'Location', and each attribute of this class would contain the columns that are in the table; for location, the attributes would be id, continent, state, country, postcode. This mapping allows information from the database to be passed through the backend of MGTdb and to the user in the frontend.

The views are python, backend files that execute the backend logic and help carry the model data and render the templates based on a user's request (Django, 2023). This happens due to the user input being

sent to a Django controller, which then sends a request to a particular URL that will map to the view. The templates are HTML files that encode the information that is being displayed to the user. These files are rendered by the views which take data from the models to display to the user.

2.6.2 MGTdb, The Codebase Currently

The current source code for MGTdb exists on GitHub, a tool for managing version control and the software development life cycle (SDLC) (Github Docs, 2023). **Figures 2.15** describes the overarching structure for the MGTdb codebase and the contents of the root folder in the MGT platform.

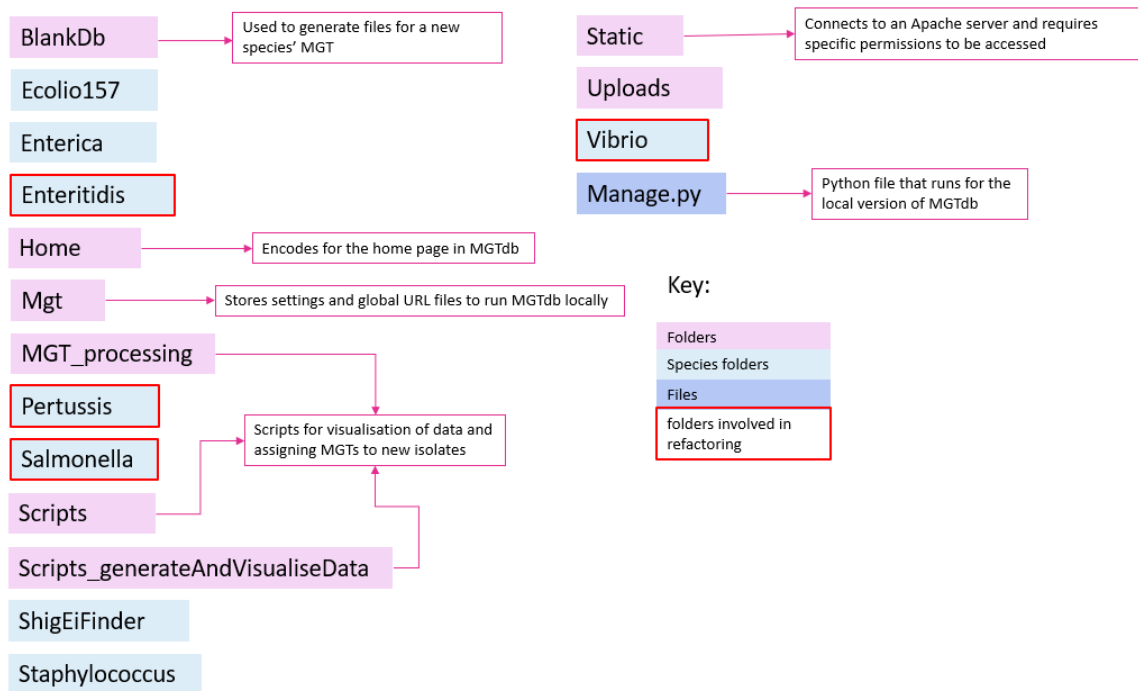


Figure 2.15: Codebase overview for non-species and species folders in the root directory of the project. Some of the species' folders were integral to this project which are outlined in red boxes.

Each species folder contains a model, view, template, template tags and static folder (**Figure 2.16**). The files that exist in the first part of the species folder include the app, admin, tests, and URLs file. While the admin and tests files are virtually empty, the app file creates a config for the species and the URLs file contains all the URLs for that species that map to a view that exists in the views folder. The URLs file in each species is identical in paths but are stored separately due to them being routed by the URLs page in the Mgt folder in the root directory.

The model folder stores python classes that map to the species' database. Each class represents a table from the database species and the attributes in each class represent a column from that table. Each species model folder, while having the same files stores different models that map according to each of the organisms' database.

The views folder stores each potential view that is mapped with one of the URLs in the `urls.py` file. Each view helps carry out a particular function in the platform such as navigating to a list of isolates in the species with assigned MGTs. The views folder also contains 3 additional folders: `ajax`, `FuncsAuxAndDb` and `cbv`. The `ajax` folder helps send requests and receive responses between the client and server and acts similarly to an API. The `FuncsAuxAndDb` folder contains the helper functions that support each view and the `cbv` folder contains class-based views. These views typically require user authentication for access.

The templates and template tags folders contain the html code for the platform. The views mentioned previously render these html files by carrying data from the models and outputting them into these template files. The template tags folder contains code for the common sections of the platform, such as the header and navigation tabs in each page. These are identical across all the species folders and don't have any hardcoded organism names in them.

The static folders contain the styles of each page and asynchronous functions that control the template rendering. These asynchronous functions are written in JavaScript, and the style files are written in CSS.

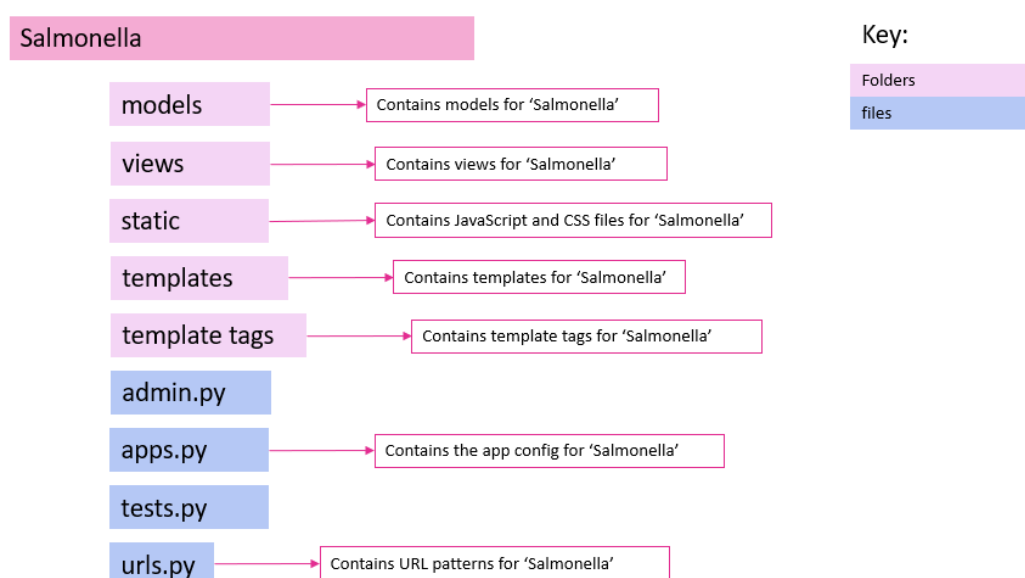


Figure 2.16: Codebase overview for a species-specific folder. The `admin.py` and `tests.py` files were empty to begin with.

2.7 MGTdb read-the-docs Documentation.

The documentation for MGTdb is hosted on read-the-docs; a software tool for hosting and versioning documentation for other applications (Read The Docs, 2023). This documentation is embedded into the MGTdb website and is in a separate repository for updates. The read-the-docs documentation contains

the following: information on the MGT technique itself, MGTdb website features, local generation of allele files to use for MGT, instructions for locally installing the MGTdb web application code, what the database schema looks like for one species and how the MGT analysis pipeline works to assign MGTs to uploaded isolates. An example of what this documentation tool looks like can be seen in (Figure 2.17).

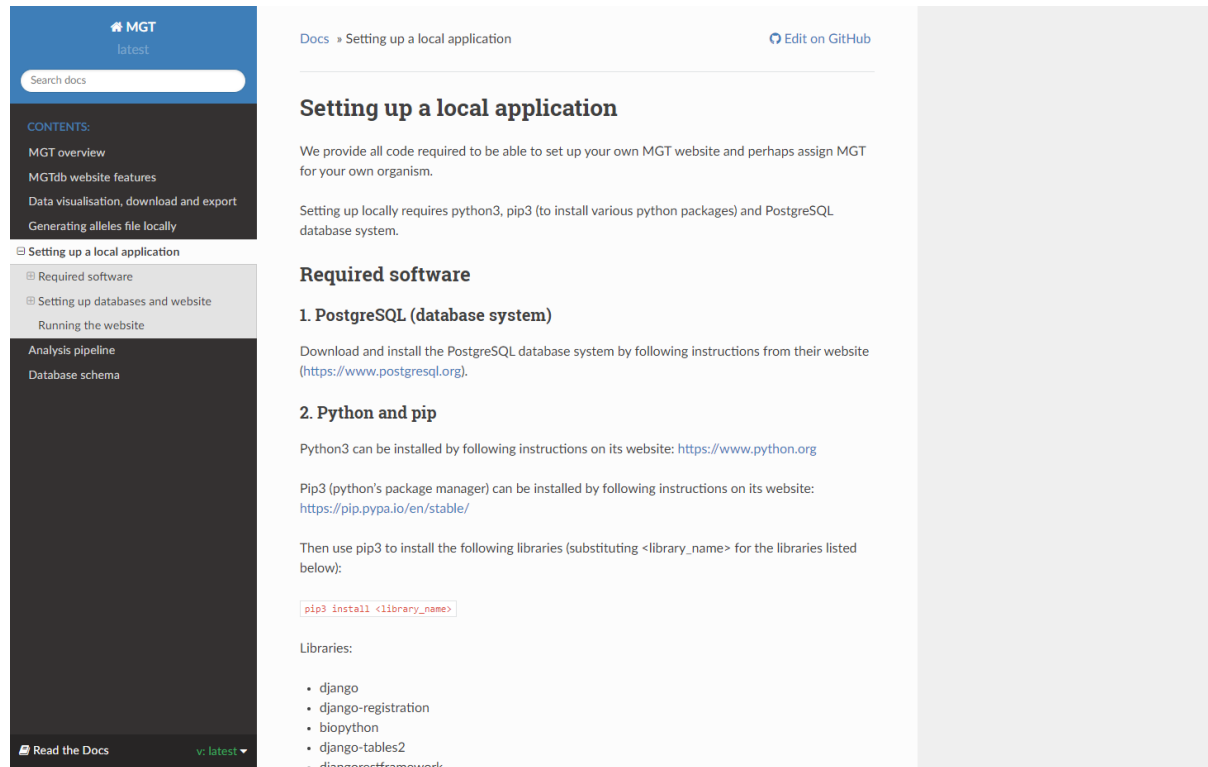


Figure 2.17: MGTdb read-the-docs interface on the page for setting up a local application. These instructions pertain to downloading the MGTdb web application code so that a user can generate the same visualisations and summaries provided by MGTdb for a species of their own interest.

2.8 Local Machine Installation of MGTdb

The current MGTdb hosts MGTs for a limited number of species. Given the large number of bacterial species and rapid generation of large amount of genome data, it is anticipated that MGTdb is applicable to any bacterial species. The web application code has been made publicly available, so that people can use the MGTdb package to develop and host their own MGT schemes for a species of interest. The code is available in the MGT-local GitHub repository (Github Docs, 2023). The contents of this repository include: the MGTdb code minus the specific models for a database, some examples of inputs required to set up a new database and a setup script that manipulates the inputs to create the models and populate the related database. An outline of this is visible in **Figure 2.18**.

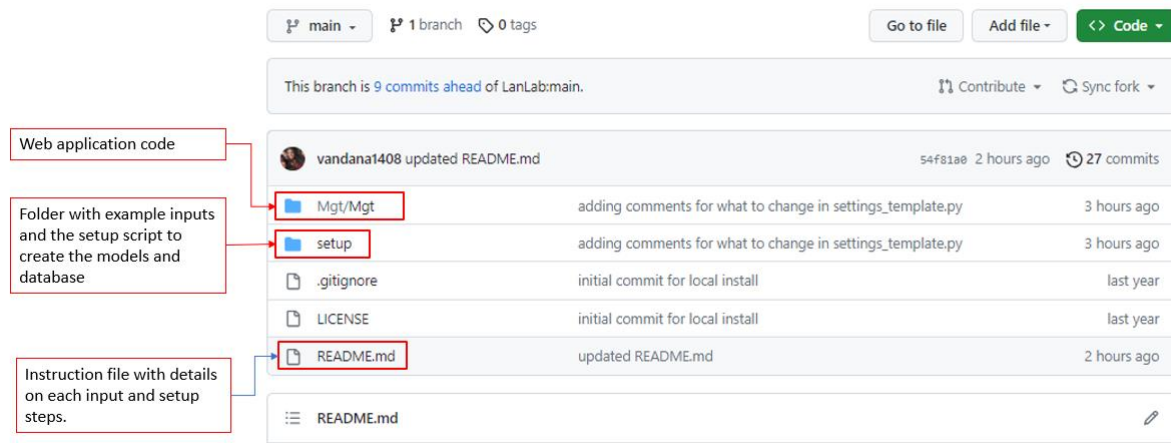


Figure 2.18: Structure of the MGT-local repository. The Mgt folder contains the web application code and an example of blank models. The setup folder contains example inputs, the runtime environment file and the setup script that creates the models and database for a new species. The README.md file contains instructions on setup.

2.8.1 The Input files

In MGT-local, the input files required help create setup files which are used to populate and create the Postgres database for the species of interest. The reference genome in ‘. fasta’ format is the first input required for setting up the database, Then, the ‘locilocations.txt’ file is another requirement; a tab delimited file with columns for: locus id, start position in genome, end position in genome, whether it’s a positive (+) or negative (-) strand and the chromosome number. A schemes folder is also required, where each file is a different MGT level, and the contents are a list of loci that will be used in that level. All three inputs examples are provided in **Figure 2.19**.

1

geneA	1	1920	+	1
geneB	1917	2609	+	1
geneC	2606	3403	+	1
geneD	3410	4171	+	1

2

```
>genome
AGAGATTACGTCTGGTTGCAAGAGATCATGACAGGGGGAATTGGTTGAAAATAAATATATCGCCAGCAGCACATGAACAA
GTTTCGGAATGTGATCAATTTAAAAATTTATTGACTTAGGCGGGCAGATACTTTAACCAATATAGGAATACAAGACAGAC
AAATAAAAAATGACAGAGTACACAACATCCATGAACCGCATCAGCACCACCACCATTACCACCATCACCATTACCACAGGT
AACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAGCCCGCACCTGAACAGTGCGGGCTTTTTTTTCGACCAGAGA
TCACGAGGTAACAACCATGCGAGTGTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTCTGCGTGTGCGG
ATATTCTGGAAGCAATGCCAGGCAAGGGCAGGTAGCGACCGTACTTTCCGCCCCCGCGAAAATTACCAACCATCTGGTG
GCAATGATTGAAAAAACTATCGGCGGCCAGGATGCTTTGCCGAATATCAGCGATGCAGAACGTATTTTTCTGACCTGCT
CGCAGGACTTGCCAGCGCGCAGCCGGGATTCCCGCTTGACCGGTTGAAAAATGGTTGTCGAACAAGAATTCGCTCAGATCA
AACATGTTCTGCATGGTATCAGCCTGCTGGGTGAGTGGCCGGATAGCATCAACGCCGCGCTGATTTGCCGTGGCGAAAAA
ATGTCGATCGCGATTATGGCGGGACTTCTGGAGGCGGTGGGCATCGCGTCACGGTGATCGATCCGGTAGAAAAATTGCT
GGCGGTGGGCCATTACCTTGAATCTACCGTTGATATCGCGGAATCGACTCGCCGTATCGCCGCCAGCCAGATCCCGGCCG
```

3

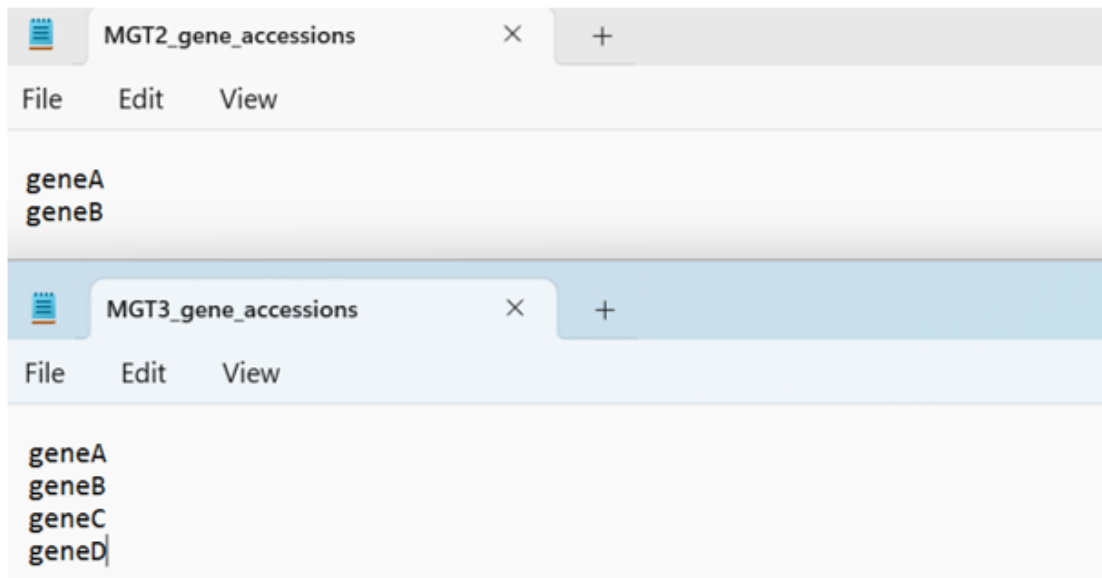


Figure 2.19: (1) Example for the loci locations file. (2) Example for the. fasta reference genome. (3) Example contents of the schemes folder, the loci are listed in each file using the IDs from the loci locations file. These must be consistent.

2.8.2 MGT-local setup process

In the setup folder, there are a few steps for creating the models associated with a user's species of interest and creating the database that is used by MGTdb for that organism. The instructions for this process are provided in the README mark down file, the first file a new user would see when they open this GitHub repository. It contains all the information regarding necessary inputs and the

instructions for setting up the local application. One of these steps involves changing variables within multiple files such as the settings configuration, global URL configuration file and the setup script file. After this, the user can run the setup script to generate their models and database to connect to the application (**Figure 2.20**).

The setup script starts by dropping the existing database if it's there and recreating it in PostgreSQL commands. After this, the program takes the input files and makes extra setup files that help create and populate the tables in the database as well as make the associated models. From here the different tables have their respective models and tables created. A scripts folder in the main 'Mgt' folder of this repository handles the logic to populate the data and write to each model's file to create a model before performing Django migrations (Django , 2023), which propagate the changes made to the models to match the database schema.

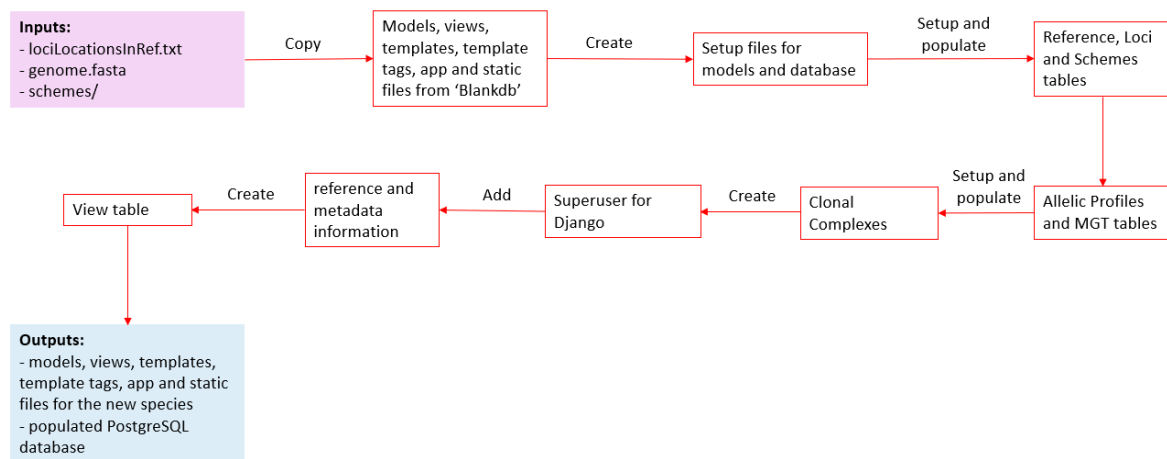


Figure 2.20: Flowchart depicting the steps of the setup script in MGT-local. Inputs are the loci locations file, reference genome and schemes folder. The outputs are the models for the species as well as a populated PostgreSQL database that is connected to the Django application.

Chapter 3

Challenges and Aims

3.1 Challenges

With the potential that MGT and MGTdb have, a long-term goal for the platform would be to store the STs for many species one day and continue further development on the platform. Currently when developers for MGTdb need to change a feature or fix a bug in the platform, they need to make the change in at least four different places, because there are at least four different species recorded in the MGTdb, and each change is virtually the same (**Figure 3.1**). If more species are added into the MGTdb platform and continue the further development alongside the increased number of species, this would make the software maintenance a very repetitive and time-consuming process.

Currently, when a new species is added to the MGTdb for users to view STs, a script is run to use the 'BlankDb' and refill all the necessary information for the new species, including models, views, and templates. However, this will simply create more files in the codebase and increase the repetitiveness of resolving bugs and implementing new features to MGTdb. The duplication in the MGTdb codebase, is carried into the MGT-local repository thus making the installation process of MGTdb locally inefficient.

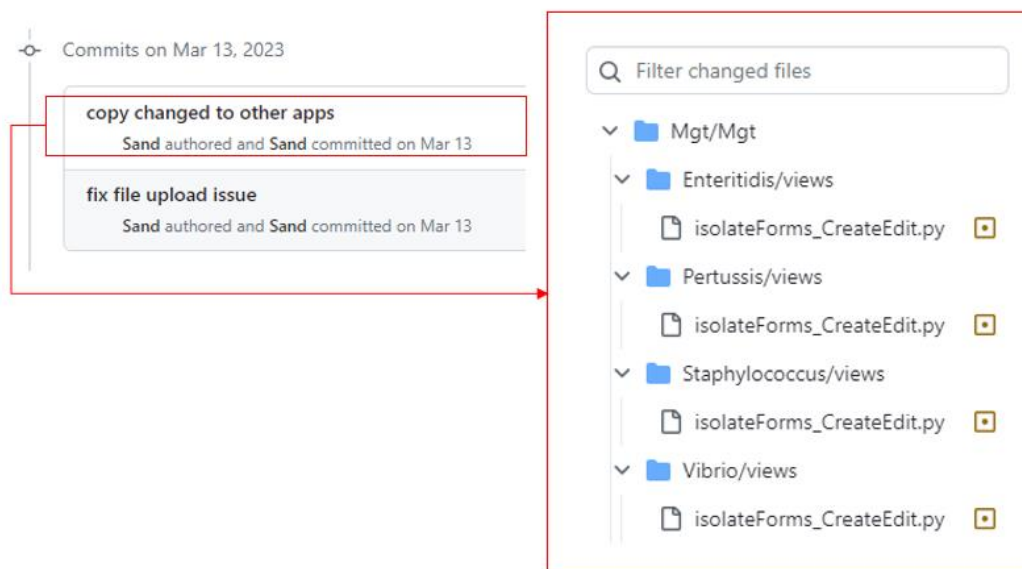


Figure 3.1: Commit history in GitHub for the MGTdb. There were two main commits made to resolve the issue. In the first commit, the issue is resolved for one of the species and in the next commit, the change that was made for the first species is made in the other species folders. In total 5 files needed changes to completely resolve this bug in the program.

3.2 Aims

The first aim of my project was to refactor the MGTdb codebase. This refactoring intended to change the code that runs MGTdb without impacting the functionality of the program. Instead of having separate species codebases that contain models, views, and templates, individually, the MGTdb codebase will contain models, views, and template folders directly in the root directory (**Figure 3.2**). Although the model files are unique to each species, the views and templates contain many similarities that are discussed in **Section 2.7.3**. These files were merged-in since the actual code contents only differ by the ‘species’ that the view is rendering for, or the template that is being rendered.

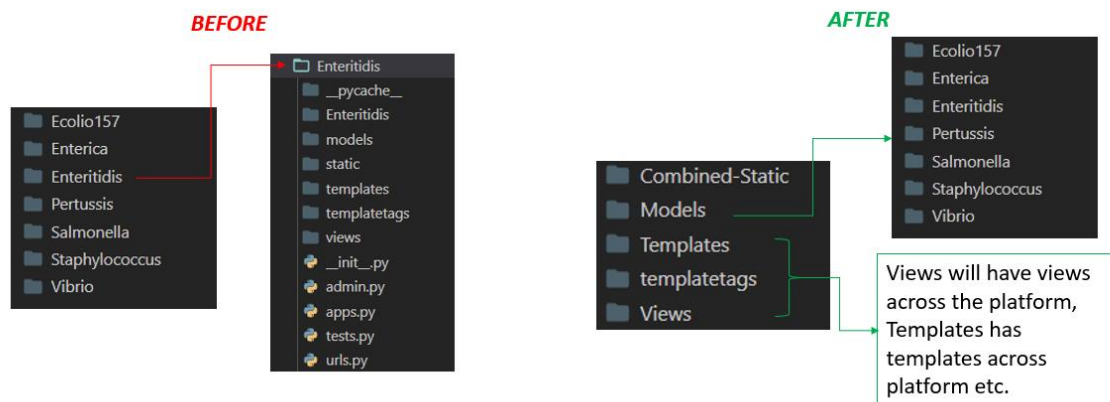


Figure 3.2: Conceptual diagram for the first aim of this project. This diagram only shows the folders that will be affected by the refactoring process. The Combined-Static folder should not be confused with the Static folder that currently presides in the root directory.

After completing the first aim, the codebase contents in MGT-local was outdated for the local installation. Therefore, the second aim was to simplify and update the installation of MGTdb on a local machine by moving the refactored code to the MGT-local repository and making the installation process more efficient for a new user. The read-the-docs and README.md in MGT-local also need to be updated for this to work.

Chapter 4

Refactoring the MGTdb Codebase

In this chapter, the methods, and results of refactoring the MGTdb codebase are described. **Section 4.1** covers the approaches used for refactoring and the testing plans or quality checks that were established to ensure that the code changes made didn't impact the functionality of MGTdb. **Section 4.2** outlines the outcomes of refactoring, including the new structure of the MGTdb codebase and the outcomes of the quality checks put in place before code changes.

Since the codebase follows the MVT architecture, the main sections of code that require refactoring were URLs, views, templates, and static files. The models don't need to be refactored since the databases are unique to each species.

4.1 Methods

The overall methods for refactoring the MGTdb codebase looked like this:

1. Setup testing plan by taking screenshots of various application pages before refactoring and writing some automated tests to ensure that the functionality of the web-application did not change after refactoring. This is described in **Section 4.1.1** below.
2. Combine the species-specific URL files into one file. This is described in **Section 4.1.2** below.
3. Combine the species-specific views into one file. This is described in **Section 4.1.3** below.
4. Combine the species-specific templates and static files. This is described in **Section 4.1.4** below.
5. Retake the screenshots throughout the application and rerun the automated tests to ensure that the program remained its original functionality. This is described in **Section 4.1.5** below.

After each step 2 to 4 some preliminary testing was done to ensure that the functionality was not affected. If not affected, that component was marked as being refactored.

4.1.1 Setup of Testing Plan Before Code Changes

This section details the testing plan. The tests were set up prior to beginning the refactoring process. It involved two main processes:

1. Screenshots of various features across the different species.
2. Automated tests.

As previously mentioned in **Section 2.6.2**, the test files in each species folder were empty. However, these files did import the Django.test module, which allows the use of the TestCase class to test any python

code (Django, 2023). Tests were written to validate a URL resource's ability to map to a view and render the correct template with information (**Figure 4.1**).

```
class TestViews(TestCase):
    databases = { 'default', 'salmonella' }

    def setUp(self):

        # set up models
        self.Salmonella_reference = Reference.objects.create(
            # salmonella reference details
        )

        # set up client and urls
        self.client = Client()
        self.url_index = reverse('Salmonella:index')
        # ... more urls

    def test_index_components(self):
        response = self.client.get(self.url_index, using='salmonella')
        self.assertTemplateUsed(response, 'Templates/index.html')
        self.assertContains(response, '/salmonella/')
        # ... Continue making assertions for common details in the template page
```

Figure 4.1: Example of code to test for the index components for the URL resource salmonella/. The setup function is used for the entire TestViews class and the test_index_components(self) method carries out the logic to test for this view.

The manual methods of testing involved screenshots throughout the application of various pages and recording them in different folders on my computer, to obtain high resolution in the images (**Figure 4.2**). The screenshots were useful for recording aesthetic information about the platform and a checklist for the database information and steps taken to record the screenshots was used. These are available in the **Supplementary Data (Frontend Screenshots)**.



Figure 4.2: Image of folder containing some screenshots of the Salmonella section of the platform, before the refactoring began, one example can be seen in more depth for the 'Isolate Details' of Isolate SRR155304.

4.1.2 URL Re-routing

To run MGTdb on a local machine, there are settings and global URL files in the 'Mgt' folder in the root directory (**Figure 2.19**). This file contained the lists of URLs that were supposed to be included in the entire application and were grouped by the species they belonged to (**Figure 4.3**). In each line that contains the contents for the global URLs, the file that contains the URLs is added to the overall list with the species name associated with these URLs. This is repeated for however many species are in the MGTdb (4 in this refactoring process). The contents of the species-specific URL files contained identical suffixes, which meant that these file contents were identical to one another.

```

"""
Contents of Mgt.urls
"""

from django.urls import re_path as url, include
# import other modules

urlpatterns = [
    # url(prefix value, list of urls to include),
    url(r'^[sS]almonella/', include(('Salmonella.urls', 'Salmonella'))),
    url(r'^[eE]nteritidis/', include(('Enteritidis.urls', 'Enteritidis'))),
    # import rest of urls
]

```

Figure 4.3: Example of global patterns including the species names and their namespaces to be added to the URL patterns. In the example for Salmonella, the include function takes the list of URLs stored in the file with the path Salmonella/urls.py and adds them to the global list with the app namespace 'Salmonella'. Salmonella.urls and Enteritidis.urls have the same contents.

To combine these URL files into one, the organism names were extracted from the variable RAWQUERIES_DISPLAY in the Mgt/settings file, which is a dictionary of species and any additional values that may be required for the raw queries that are used for the views. RAWQUERIES_DISPLAY is looped over to dynamically update and add the species-specific URL paths to the global URLs list. This code logic was stored in a file 'MGTdb_shared/urls.py' (**Figure 4.4**). The urlpatterns that are initialised and added to in 'MGTdb_shared/urls.py' are then added into the global list of URLs across the entire application (**Figure 4.5**).


```

"""
MGTdb_shared/urls.py
"""

# import modules

urlpatterns = []

for org in settings.RAWQUERIES_DISPLAY
    1. define list of url patterns to belong under org
    2. add this new list to the overall urlpatterns of the
        program with org as a prefix to the list of urls

```

Figure 4.4: Pseudocode of contents in MGTdb_shared/urls.py to show that URLs were updated dynamically and added to the 'urlpatterns' variable in the file.

```

"""
Contents of Mgt.urls
"""

from django.urls import re_path as url, include
# import other modules

urlpatterns = [
    url(include(('file.containing.urls', 'prefix of these urls')))
    url(include(('MGTdb_shared.urls', 'MGTdb_shared'))),
    # import non-species specific urls
]

```

Figure 4.5: updated version of the global URLs file, after refactoring the species-specific URL files.

4.1.3 View Parameterising

Any species-specific URL list maps to their respective views in each species folder. For example: the URLs listed in 'Salmonella/urls.py' previously mapped to views stored in 'Salmonella/views'. Each species-specific view contained almost identical logic, aside from a hard-coded organism name throughout these code files. To make the views more generic, the species name was added as a parameter to each view and the helper functions.

The views import the models into their code files to manipulate and transform to pass into templates. However, the models to be imported are also dependent on the species that is being accessed, meaning that is another hard-coded aspect of the views. To generalise this, I used the python module ‘importlib’ (Python, 2023) and created a generic function in each view file and helper file, to dynamically import the models based on the species name. This function is then called at the start of every view and helper function to be used (**Figure 4.6**).

```

"""
View Parameterising
"""

'1'
def page(request, org):
    model definitions = getModels(org)
    # rest of the view function

'2'
# dynamically import modules with this function
def getModels(org):
    models = importlib.import_module(f'{org}.models')
    2. Add model definitions that are used in the view

    return the model definitions

'3'
def getIsolates(param1, param2, ..., org):
    model definitions = getModels(org)
    # rest of the function

```

Figure 4.6: Pseudocode of view parameterising based on the steps taken. (1) add ‘org’ as a parameter or each view and helper function for the views, (2) create the getModels function to dynamically import models from a given species and (3) call the getModels function at the start of every helper function.

In the helper functions, there were some raw SQL queries that helped retrieve information directly from the database. Some of these queries are specific to the species, particularly in the isolate list table from **Section 2.5.2** in **Figures 2.5, 2.6, 2.7 and 2.8**. *Vibrio Cholerae* and *Bordetella Pertussis* have extra columns in their view table to display further information about the isolates that's specific to them. To make these specifications more abstract in the view files, the 'RAWQUERIES_DISPLAY' constant was defined in the Mgt/settings file. This contains a dictionary with each species was defined as a key and the value was the additional parts of the query string that needed to be included in the table of isolates (**Figure 4.7**). The key value for each species is concatenated to the raw queries as needed (**Figure 4.8**).

```
# from the settings file
RAWQUERIES_DISPLAY = {
    'Species': 'columns for raw queries'
    'Salmonella': '',
    'Enteritidis': '',
    'Vibrio': ', i.VC2210',
}
```

Figure 4.7: 'RAWQUERIES_DISPLAY' variable contents. *S. Typhimurium* and *S. Enteritidis* have the default columns in their database, whereas *Vibrio Cholerae* and *Bordetella Pertussis* have species specific information which is represented in the query string.

```
# from rawQueries.py in the FuncsAuxAndDb folder of views
def getIsolates(isoSearchStr, islnIds, ..., dir, isMgt9Ap, searchType, isFirstSearchType, org):
    View_apcc, Location, Isolation, Isolate, User, Tables_cc = getModels(org)

    isolates = []

    displayStr = 'i.id, i.identifier, ..., i.mgt1, v.*, iM_l.*, iM_i.*' + RAWQUERIES_DISPLAY[org]

    # rest of code logic
```

Figure 4.8: Example use case for RAWQUERIES_DISPLAY in the raw queries used throughout the platform.

After these steps were completed and some preliminary tests were done. I moved the edited 'Salmonella/views/' folder into MGTdb_shared/views/ and delete the existing view folders for the other species in the platform, thus clearing up unnecessary code files.

4.1.4 Template and Static File Parameterising

Previously, each view file rendered a particular template that was specific to the species through returning a render object in the view. The render object takes a file path for the template to be rendered and the context data, which contains any values from the models that were manipulated in the views. This is how the templates display database specific information to the user.

The context data is a dictionary that contains transformed data from the views to be displayed in the templates. The templates then pass in the key name from the context dictionary where applicable, making this part of the logic identical for any species. The differing aspects of the templates were from the hardcoding of the species name in each file to be navigated to whether it was in hyperlink variables or simple text for the page.

To resolve this, the organism's name was added to the context data, eliminating the need to hardcode the organism's name in the templates. Most of these replacements were for buttons that contained hyperlink redirects to other pages on the website, so the organism parameter had to be reformatted into a string using 'stringformat' (Template Tags and Filters, 2022) before being passed as a parameter for these reverse URLs (**Figure 4.9**).

```
...
Template Parameterising
...

# from isolateDetail.py file in the root of views folder
context = {"isolate": isolate,
           "serverStatusChoices": list_serverStatus, ...,
           'organism': org}

return render(request, 'Templates/isolateDetail.html', context)

# from isolateDetail.html in the templates/Templates folder
<a href="{% url organism|stringformat:'s:isolate_edit' isolate.id %}"
class="btn btn-outline-warning">Edit</a>
```

Figure 4.9: Pseudocode for parameterising the templates to remove the hardcoded species name in the URL redirects. The key from the context dictionary 'organism' is used as a variable throughout the templates to represent the organism that the view logic made the transformations for.

The JavaScript files were mostly identical to one another with a few ajax call differences that hardcoded the organism's name into these functions. However, the main task for static files was removing the species name as a prefix on these JavaScript functions from the templates. To re-parameterise the ajax calls, I added the organism parameter which could be passed in from the template's context data into the JavaScript function where mentioned (**Figure 4.10**).

```
// (1) from IsolateTable.html in templates
<button ... onclick="javascript:getTimeStCount({{tabAps}}, {{tabCcs}},
'{{organism}}');"> Load data </button>

// (2) from isolateList.js in static files
window.getTimeStCount = get_timeStCount;

// (3) from graph_timeOrLocStCount.js in static/modules files
function get_timeStCount(tabAps, tabCcsOdc, org){
    // some logic

    ajaxCall('/') + org + '/timeStCount', {}, handle_timeStCount);
}
```

Figure 4.10: Pseudocode for parameterising static files. The JavaScript function in the html line previously had the species name as a prefix to the function title (1) but this was removed since the code logic is the same across all species. This was also done for the JavaScript files that were renaming the functions to be imported to the .html files (2). The organism is passed as a parameter to the function and added to the respective JavaScript function (3).

4.1.5 Validation and Testing After Code Changes

To validate that the code changes did not impact the functionality of MGTdb, screenshots of each page for three of the species were taken to ensure that the aesthetics and values for each page were the same as before. These new screenshots were compared to the ones taken previously before refactoring began. Additionally, the Django test scripts were run again to validate that the view-to-template rerouting was maintained, since there were changes to the file locations after refactoring. The screenshots from before and after the code changes can be found in the **Supplementary Data (Frontend Screenshots)**. Once it was confirmed that the functionality was the same as what is described in **Section 2.4**, the refactored code was moved into HGTdb, the testing server for MGTdb.

4.2 Results

Before refactoring, the movement of data through the application was isolated to each species. For example, all the models for *Salmonella enterica* serovar Typhimurium (*Salmonella*) only went through the *Salmonella* views and these views only rendered the *Salmonella* templates (**Figure 4.11**). However, after refactoring these models are all passed into the same view files and passed into the same templates (**Figure 4.12**). The refactored code is currently in HGTdb, the testing server and other developers can use this code for their feature development or database additions into MGTdb. The refactored code is available in the **Supplementary Data (Refactored MGTdb Codebase)**.

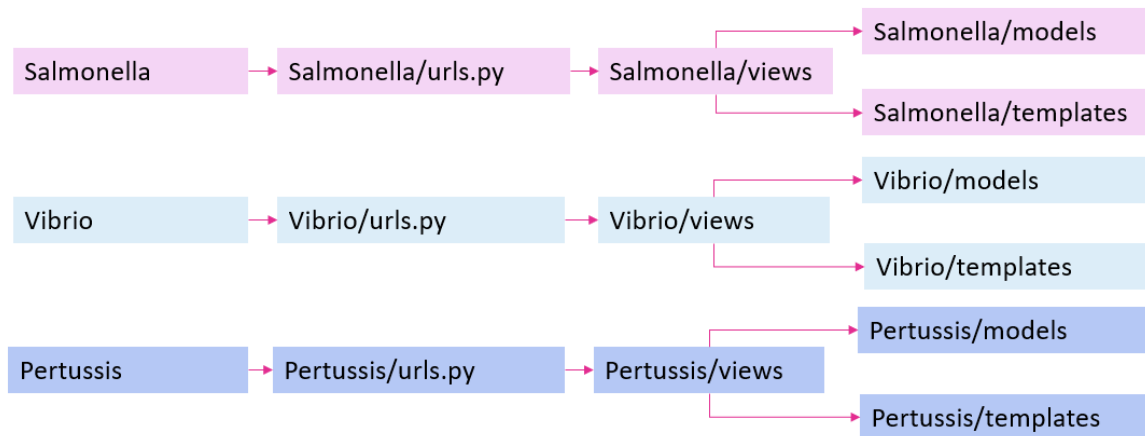


Figure 4.11: Flowchart of pathways for *Salmonella*, *Vibrio* and *Pertussis* databases, before refactoring. The separate URL files cause the species databases to act as separate codebases, despite being in the same Django application with similar contents in the URLs, views, and template files.

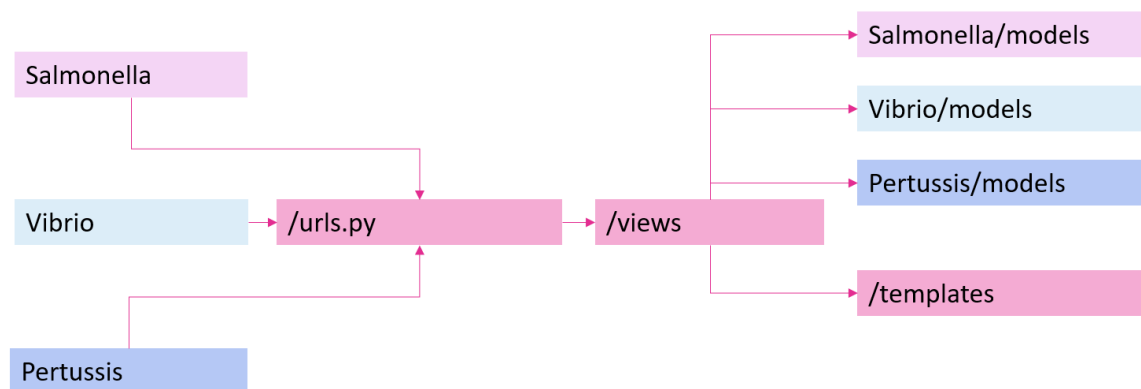


Figure 4.12: Flowchart of pathways for *Salmonella*, *Vibrio* and *Pertussis* databases, after refactoring. The URLs, views and templates are all in one codebase, with the model information passed into this. The models to be used in each view are based on the value of the organism parameter that was passed into it from the dynamic URLs. This then gets added to the context data for each of the templates which also contain species specific information.

The combined URLs, views, templates and template tags and static files were moved to the MGTdb_shared folder and an app configuration was added to this folder so that this combined codebase could still communicate with the rest of the application (**Figure 4.13**). This also means that the INSTALLED_APPS variable in Mgt/settings.py also contains MGTdb_shared as an app now (**Figure 4.14**).

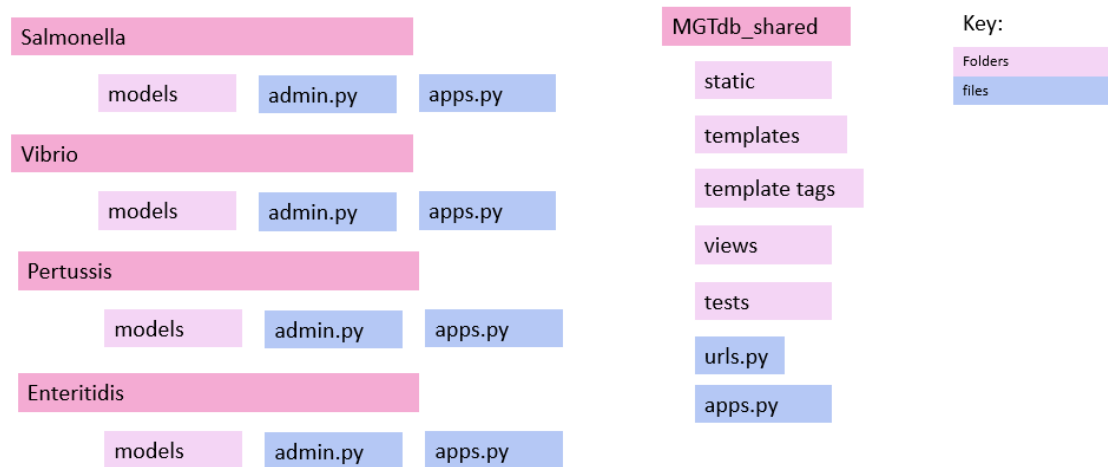


Figure 4.13: Codebase contents after refactoring the MGTdb codebase.

(1) from the settings file

```
INSTALLED_APPS = [
    'django_tables2',
    'Home',
    'MGTdb_shared',
    'more apps',
    ...
]
```

(2) App config in MGTdb_shared
from django.apps import AppConfig

```
class SpeciesConfig(AppConfig):
    name = 'MGTdb_shared'
```

Figure 4.14: (1) change to the INSTALLED_APPS variable in the settings file. INSTALLED_APPS must always contain the MGTdb_shared app. (2) App configuration in MGTdb_shared/apps.py.

Chapter 5

Simplifying and Updating the Local Installation of MGTdb

After refactoring the codebase, the MGTdb_shared folder needed to move into the MGT-local repository, remove the views, templates, template tags and static folders from the 'Blankdb' example folder and update the setup script's commands to accommodate for the new changes. The corresponding documentation also needed to be updated. **Section 5.1** covers the methods used to simplify and update the setup and installation process of MGTdb locally. It details the different approaches to simplifying the setup and what decision was made in the end. **Section 5.2** entails the results from updating the setup script and documentation, showing the changes to the MGT-local repository and read-the-docs repository.

With implementing the MGTdb_shared folder, some of the previous steps that were required for setting up a local application such as, coping and replacing the default species folder, with the new species-name and settings, can be simplified. More steps for the MGT-local installation like this one can be enhanced for a new user so that installing and setting up MGTdb on their own servers is much more doable.

5.1 Methods

After the MGTdb codebase was refactored, the new MGTdb_shared code needed to be moved to the MGT-local repository. This meant that the setup and installation steps for MGT-local needed to be updated to accommodate for the refactored code as well. There were two main parts to this process:

1. Simplify and update the MGT-local setup process to account for the refactored code. This is described in **Section 5.1.1** below.
2. Update all documentation for the setup process to reflect changes. This is described in **Section 5.1.2** below.

5.1.1 Simplify and update the setup and installation process.

The first part of updating the setup and installation process was running the previous setup steps with a made-up database and using the provided example inputs. This helped me identify which sub-processes were no longer required for the refactored code and make slight changes to the script to accommodate this.

The next step was to Dockerise the application and databases to simplify the installation and setup process. Docker is a software tool that allows users to put their application into containers (Docker docs, 2023). Containers are lightweight virtual machines that contain everything that is required for a program

to run (Docker docs, 2023). This includes: the application code and runtime environments that the code can be run in (**Figure 5.1**). The main benefit of containerising MGT-local is that the local setup steps that involve downloading Python, Pip and PostgreSQL could be disregarded because the containers would contain all the necessary libraries, system tools, code, and runtime environment. However, some difficulties were encountered while trying to Dockerise MGT-local, making it difficult to Dockerise the platform.







Name	Image	Status	CPU (%)	Port(s)
 dmdb		Exited	N/A	
 web-1 304e740d9fc3 	dmdb-web	Exited (137)	N/A	8000:8000 
 db-1 dfa120aa7178 	postgres:13.1-alpine	Exited	N/A	

Figure 5.1: Screenshot of Docker Desktop that illustrates how containers theoretically work for an example Django application. Here, there is a container for the web that contains the web application code (views, templates etc.). The second container is for the database which is running in PostgreSQL.

To setup a container for the application, the user needs to create a configuration in a ‘docker-compose.yml’ file. Since MGTdb is typically a multi-database Django application, separate containers would need to be configured in the ‘docker-compose.yml’ file for each new database that a user wanted to add (Tutorialspoint, 2023). An example of this is shown in **Figure 5.2**. Therefore, rather than simplifying a new user’s work, this would add more complexity to the setup process. Additionally, most documentation for setting up Docker with a Django application assumes the Django project is in its infancy, rather than completely built the way MGTdb is (Dockerize. The Docker Hosting, 2020). This made setting up the web container difficult.

```
# example docker-compose.yml

version: '3.8'

services:
  # web configuration
  app:
    build: .
    links:
      - salmonelladb # add the link to web for each new database created
      - vibriodb

  # first database configuration
  salmonelladb:
    image: postgres
    environment:
      PASSWORD: mypassword

  # second database configuration (each new db needs
  # a set of these lines)
  vibriodb:
    image: postgres
    environment:
      POSTGRES_PASSWORD: mypassword
```

Figure 5.2: Example docker-compose.yml file with configurations for salmonella and vibrio databases. If a user wanted to add more databases, they would have to keep changing this file on top of the other files that need to change, thus creating more work.

Therefore, I resolved to simplifying how the current setup script, which is described in **Section 2.8.2**. One aspect of the setup process I focused on was the need to update the variables in this script file, each time a new database was being added to the local application (**Figure 5.3**). Making constant changes directly in the setup script could also put a researcher at risk of accidentally creating a new bug within the code implementation.

To resolve this, I changed the script so that it takes a command line argument for an extra input file called 'setupPath.py'. This contains the variables required for the script to setup a new database (**Figure 5.4**). In the command line, the path to the setupPath.py file is an additional argument where each folder is split by a period rather than a forward slash, because this is the format in which python files import custom modules, variables, or function from other python files. The script then uses python to unpack

the variables from the specified path file so that the rest of the script's logic could unfold as previously intended (**Figure 5.5**). The setupPath.py file is also another input required by the user for the MGT-local repository.

```
#### setup script's variable assignment ####

db_name='clawclip' # CHANGE to database name
appname='Clawclip' # CHANGE to app name
dbuser='blankuser' # CHANGE to postgres user

# MORE VARIABLES TO CHANGE BELOW
# ...
# Carry out script logic
```

Figure 5.3: Pseudocode for how some variables were assigned in the setup script before changes. Each variable was hardcoded into the script and if a user wanted to add another database, they would need to change the assignments to these variables again.

```
### setupPath.py file example ###

# CHANGE Database name
DB_NAME='clawclip'

# CHANGE App name (usually the database name with a captial letter)
APPNAME='Clawclip'

# CHANGE Species name that you want displayed with the page.
SPECIES = '<i> Cute Beige Claw Clip </i>'

## more variables assigned here.
```

Figure 5.4: Example of setupPath.py using the made up 'Clawclip' database. setupPath.py contains python variables which are taken into the command line so that the contents of this file can be unpacked into separate variables and run successfully, in the setup script.

```
#### setup script's variable assignment ####

# takes first argument of the shell command (the input file) and assigns it to SETTINGS
SETTINGS="$1"

# Some variable updates
export db_name='$(python -c "from $SETTINGS import DB_NAME; print(DB_NAME)")'
export appname='$(python -c "from $SETTINGS import APPNAME; print(APPNAME)")'
export dbuser='$(python -c "from $SETTINGS import DB_USER; print(DB_USER)")'

# Continue updating more variables and then
# Carry out logic of setup script
```

Figure 5.5: Pseudocode for how some variables were assigned in the setup script after code changes. The first code line in the program assigns the first argument of the command line to the settings variable. After this, the previous variables are now respectively assigned to the value that it is associated with from the setupPath.py file.

5.1.2 Update MGT-local and read-the-docs Documentation.

After the setup script was simplified and updated, I updated the documentation of instructions in MGT-local and the read-the-docs documentation for installing MGTdb on a computer. This involved changing three main areas: the MGT-local README.md file, the read-the-docs documentation page on ‘Setting up a local application’ and adding comments to any new changes made throughout files used in the setup process.

In the MGT-local README.md, I updated the setup script command that is run to set up a local application and removed the user’s need to change variables in the global URL configuration as well as the setup script. Additionally, there are less variables to change in the settings configuration file because the pathways for the script-generated files that populate the models and database are all moved into a folder called ‘data’. I have also made a note within this settings file that a user does not need to update this but should move the folder once the setup process is completed. The MGT-local README.md was also changed to mention the new input and makes a clear reference to ‘Clawclip’ as a prime example of what changes in the settings file should look like to a new user.

In the read-the-docs documentation, the main section I needed to update was instructions for setting up a local application. I changed this section to incorporate the new way to run the setup script and refer to the fact that most changes will need to be made in the settings configuration file. Additionally, within the settings file, and the example setupPath.py input, I made the inline comments for changes to the file more explicit for a user, providing examples of what the changes should be. The ‘Clawclip’ database is a configured example that the users can follow for making their changes to the application.

5.2 Results

The refactored code is now part of the MGT-local repository and changes to the setup script were made accordingly so that it only creates new models and the database for a new application (**Figure 5.6**).

Moreover, the setupPath.py has been added to the example inputs so that a user can provide this information separately, without having to manually change the setup script each time they want to add a new database. To run the modified setup script the user would have a new command line argument which takes the path of the setupPath.py file delimited with periods. **Figure 5.7** shows how it should be run based on the current working directory being the setup folder in MGT-local.

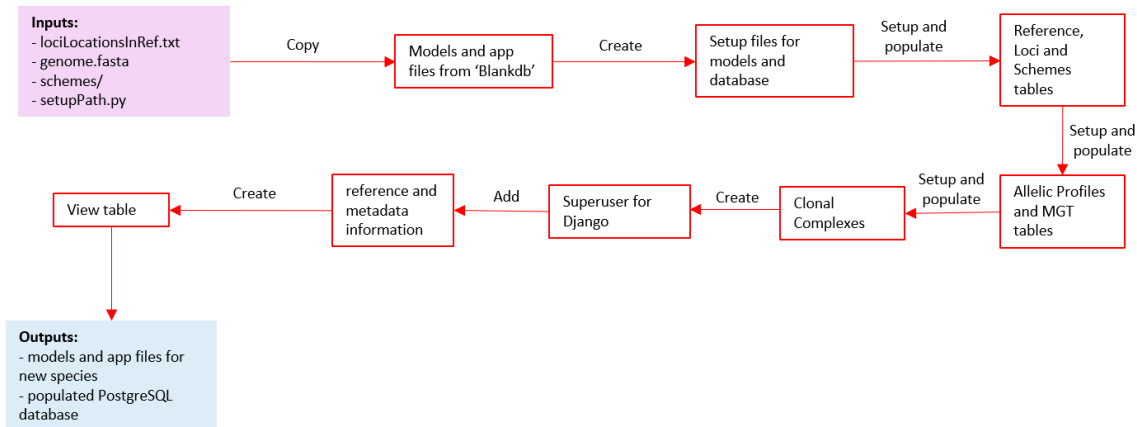


Figure 5.6: Updated process for how the setup script now runs, taking in the extra input ‘setupPath.py’ to help generate the models and database. This process no longer requires the views, templates, and template tags to be copied from ‘Blankdb’ and have its organism name changed over since the refactored code accounts for this.



Figure 5.7: Example of how to run the modified setup script.

Additionally, the read-the-docs documentation is currently updated for a local installation application as well as the README.md in the MGT-local repository (**Figure 5.8**). The line changes for the settings file and the variables for the setupPath.py file have also been accounted for in the application (**Figure**

5.9). The updated MGT-local repository is available in the **Supplementary Data (MGT-local codebase)**.

2. install dependencies

1. install [postgres](#)
2. [miniconda](#) or [mamba](#)(recommended) for environment management
3. create an environment within conda/mamba using the included yaml file (/setup/mgt_conda_env.yaml). Default name is mgtenv.

```
conda env create --file=/path/to/MGT-local/setup/mgt_conda_env.yaml
or
mamba env create --file=/path/to/MGT-local/setup/mgt_conda_env.yaml
```



3. modify settings

In the /Mgt/Mgt/Mgt folder find the settings_template.py file and rename any lines with #CHANGE comments as per comment instructions. You can also make a copy of this file and update the changes in the copy. *Clawclip* is an example of what the changes would look like.

4. run /setup/setup_new_database.ssh

use the command as follows: `./setup/setup_new_database.ssh example_inputs_2.setupPath` in command line use postgres password when prompted

5. access local mgt database site

run `python manage.py runserver --settings Mgt.settings_template` and access the website locally using host in settings file (<http://localhost:8000/> by default)

Figure 5.8: Updated MGT-local README.md documentation. This updates the ‘modify settings’ step to only change the settings file and directly mentions the ‘Clawclip’ example that is already configured in this file for users to follow as an example.

Setting up databases and website

1. The code and variables to update

The code is available at the following git repository: <https://github.com/vandana1408/MGT-local>

Clone this repository. (If you don't have git, download the code as a zip file from the repository location, and then extract it.)

To get the website running a few variables in the *settings_template.py* file need to be updated for your species of interest and some files with genome and loci information need to be provided. More information on this is provided in the README.md file located in the root directory of the codebase.

2. Setting up the databases

After making changes to the settings file, run the *setup_new_database.sh* script to setup the new databases from the setup folder as follows.

```
./setup_new_database.sh example_inputs.setupPath
```

Some prompts for a PostgreSQL password will appear throughout the course of the script, once these are entered, the program will setup the models.

Figure 5.9: Updated section in the read-the-docs documentation, that shows how to run the setup script with the new input's relative file path delimited with periods, rather than a forward slash. The updated read-the-docs documentation is available in: <https://mgt-docs.readthedocs.io/en/latest/index.html>.

Chapter 6

Discussion

This chapter will cover an overall evaluation of whether this project has been a successful one in both refactoring MGTdb and the simplification of the MGT-local installation process. **Section 6.1** is focused on the easier maintenance of MGTdb with the new codebase, how the quality checks ensured that the functionality of MGTdb was not impacted and the impacts of this codebase for ongoing projects. **Section 6.2** highlights the accessibility of MGT-local to researchers when they want to look at their own species of interest. It also covers the comprehensiveness of the new documentation within the example input files and files that need changing before running the setup script for MGT-local. **Section 6.3** outlines the further developments and future work, that is easier for developers to implement with the new codebase. Overall, this project has made further developments of MGTdb and the local installation of MGTdb more efficient, thus making the web application more readily available to researchers and helps them to draw conclusions in their findings from the MGT visualisations and summaries easily.

6.1 MGTdb Codebase Refactoring

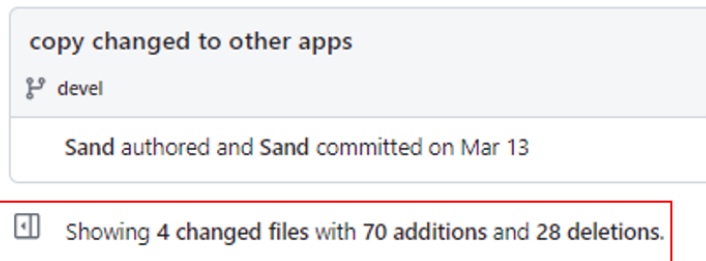
6.1.1. Successful Refactoring of MGTdb codebase

Two main factors determine the successfulness of the code refactoring:

1. Implementing new or revised code/features in a server with multiple MGT databases is more efficient and
2. The unchanged functionality of MGTdb after the codebase's changes.

The first aspect to consider for refactoring was whether the new codebase was more maintainable and makes further developments easier. As seen in **Figure 6.1**, There's a significant decrease in the number of files to change and changes itself for a bug fix or new feature development. Therefore, the new refactored codebase has achieved the goal of making features or code maintenance easier for developers of the platform.

1



2

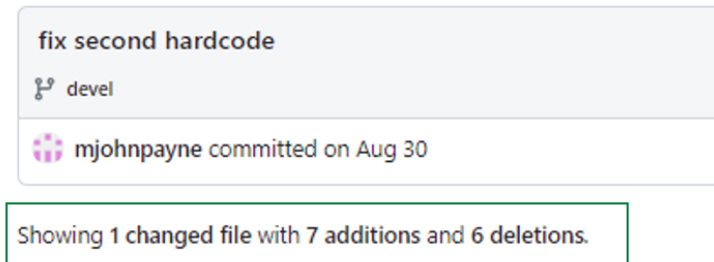


Figure 6.1: (1) Screenshot of a GitHub commit for feature development or bug fixes in the application. In this commit, 4 files had to be changes and there were 70 additions and 28 deletions. (2) Screenshot of a GitHub commit for feature development or bug fixes in the application. In this commit only one file was changed and only 7 additions and 6 deletions needed to be made.

As mentioned in the methods for Refactoring the MGTdb Codebase **Section 4.1.5**, I took screenshots throughout the application of three different species to compare to the ‘before code changes’ screenshots that were taken before any code changes. Since the database information appeared the same for the specific species’ as it would have before code changes, I determined that the code changes did not impact the previous functionality of MGTdb.

6.1.2 Impacts of Refactored code for ongoing projects in MGTdb

As referred to in the results section for Refactoring the MGTdb **Section 4.2**, the refactored code for MGTdb is currently in the testing server HGTdb. This code is accessible to the rest of the developers so that they can use it for their ongoing projects related to MGTdb.

One example is an entire Salmonella MGT database which will combine *Salmonella enterica* serovar Typhimurium and Enteritidis along with other Salmonella species into one database. While the models for this entire database need to be made, integrating them into the MGTdb no longer requires a duplication of URLs, views, templates, template tags and static files. The same steps can be applied for the new database for *Shigella Flexneri*, another ongoing project for the MGTdb.

Moreover, the portability of the refactored codebase for views, templates, template tags, URLs and static files makes this code transferrable to any new species or any ongoing developments for MGTdb.

6.2 Simplify and Update Setup and Installation Process and Documentation

Previously for the setup process of MGT-local, three file changes needed to be made: the settings configuration file, the URL configuration file, and the setup script. Each time a user wanted to add a new database to their local machine's MGTdb, they would have to update the three files in every iteration and run the script again. This process is tedious and making multiple changes directly in the setup script file could make the code susceptible to accidental bugs. Additionally, the setup script handled copying views, templates, template tags and static files for the new database, which simply adds more files for the user to deal with.

With the refactored code being transferred into MGT-local, the setup script no longer needs to handle copying views, templates, template tags or URL files. This also means that the global URL configuration doesn't need to be updated by the new user because the URL file in MGTdb_shared will dynamically create URLs for the species that are being added and this gets contained in the global URL file (**Figure 6.2**). Additionally, since the new input for creating a new database was added to the MGT-local, the setup script does not need to be updated, thus reducing the likelihood of an accidental bug being created in the script. Therefore, the updates to the setup and installation process make creating and adding a new database easier for researchers to use the web application code for MGTdb on their species of interest and use the visualisations and summaries that the web code provides.

Although, creating Docker containers for MGT-local was not successful, there are some things I learnt from the process. The tool is still very useful for Django applications and would ensure that all users had the same runtime environment when they setup and install MGT-local. However, the documentation for setting up Docker with existing Django applications still needs to be refurbished and Docker itself could be updated so that multiple databases using the same engine can be in the same container.

```

"""
Contents of Mgt.urls
"""

from django.urls import re_path as url, include
# import other modules

urlpatterns = [
    url(include(('Home.urls', 'Home'))),
    url(include(('MGTdb_shared.urls', 'MGTdb_shared'))),
    # import non-species specific urls
]

```

Figure 6.2: Example of global URL configuration in MGT-local. With MGTdb_shared URLs, the user only needs to update the RAWQUERIES_DISPLAY variable in the settings file with their species of interest and no longer needs to edit the URLs since RAWQUERIES_DISPLAY is automatically passed into MGTdb_shared URLs configuration.

A major benefit of an additional input file for the variables to pass into the setup script is that documenting what each variable is for and examples for each variable within this file is much more readable to a new user. The *Clawclip* example provided through the example inputs and settings configuration file is much clearer for a user to see what the variable updates would look like for their own species and easier to mimic. Furthermore, the *Clawclip* example is referenced in the MGT-local README.md instructions. The setup command that is used to run the setup script has also been updated in both the read-the-docs for MGTdb and the MGT-local README.md instructions.

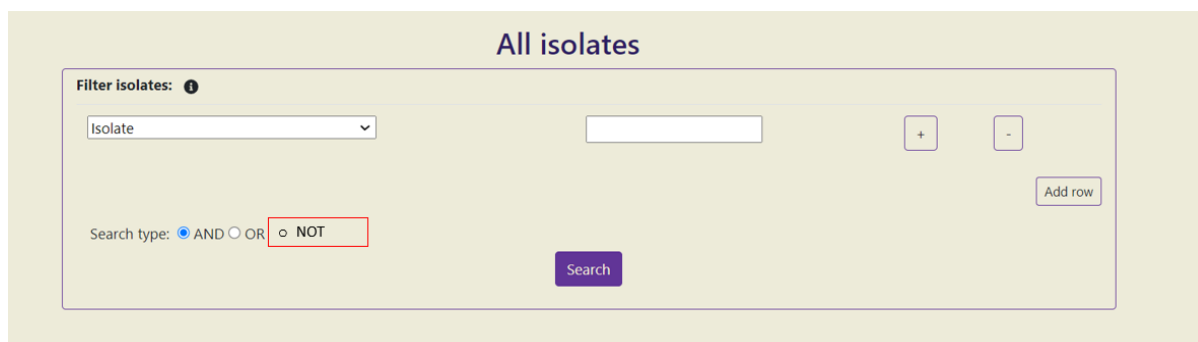
The documentation for MGTdb was updated successfully to account for the refactored code being moved to MGT-local. The instructions are more comprehensive within the example inputs and files that need to be changed and the overall setup process is much more efficient since the MGTdb_shared folder is more portable for the user.

6.3 Future Work

Refactoring the MGTdb codebase and updating the setup and installation processes for MGT-local has made future work on the web application more feasible. There are many developments that could be made around the web code itself and the process for local installation to make it that much more optimal for the long-term future. **Section 6.3.1** discusses feature specific developments; **Section 6.3.2** describes enhancements that could be made to the local installation process and **Section 6.3.3** describes the importance for finding a stable solution for hierarchical inconsistencies and implementing this into the MGTdb platform.

6.3.1 MGTdb Further Development

In terms of the MGTdb itself, having the ability to enhance the search and filter systems for all isolates in a species is much easier to implement in the application since it only needs to be implemented once. The enhancements to the search functions of all isolates for a species include searching for the ability to include ‘NOT’ as a filter option (**Figure 6.3**) for the isolate table and the ability to search isolates based on their genome type (GT), which is a combination of STs for an isolate across all the MGT-levels for that species.



The screenshot shows a web application interface titled "All isolates". Below the title is a search filter box. Inside the box, there is a "Filter isolates:" label with an information icon. Below this is a dropdown menu labeled "Isolate". To the right of the dropdown is a text input field. Further right are two buttons labeled "+" and "-". Below the dropdown and text input is a "Search type:" label with three radio button options: "AND", "OR", and "NOT". The "NOT" option is highlighted with a red rectangular box. Below the search type options is a "Search" button. To the right of the search type options is an "Add row" button.

Figure 6.3: Example of what including ‘NOT’ as a search type in the database application would look like. The red box indicates the additional search type.

6.3.2 Further enhancements to Local Installation of MGTdb

For the MGT-local installation and setup process, further optimisation of this process would make this web code more accessible to researchers who are less familiar with programming and code environments. One way to do this would be by removing any need to interact with any of the MGT-local code files altogether. This can be done by updating the setup script to automate the changes made to the settings configuration file so that the user only needs to provide inputs and the same setup command to update everything for them. Moreover, if the researcher has updated their loci selection for the MGT schemes of their species of interest, a script that removes the old database from being configured to the web application would be more beneficial to them. From these additions to MGT-local, the researcher would only need to provide their inputs in an examples folder and interact with the command line, thus avoiding any sort of disruptions in their local application installation process.

6.3.3 Fixing Clonal Clusters for resolving Hierarchical Inconsistencies

Information on Clustering and Hierarchical Clustering is available in the **Supplementary Information (Clustering and HierCC)**. Despite its issues with instability, the HierCC concept is useful in resolving hierarchical inconsistencies in MGT. Hierarchical inconsistencies occur when two isolates at a lower MGT level have different MGT STs assigned to them but have the same ST assigned in a higher MGT level. The example below demonstrates that for isolates ERR1069296 and ERR1046118, the MGT4 values are 1046 and 10 respectively, but in MGT5, the assigned ST is 9 for both (**Figure 6.4**). Clustering can be implemented on the MGT schemes by reassigning the STs based on a maximum allelic distance of 1 between the isolates to have the same MGT-CC value, which is carried out at all MGT resolutions (Payne, et al., 2020).

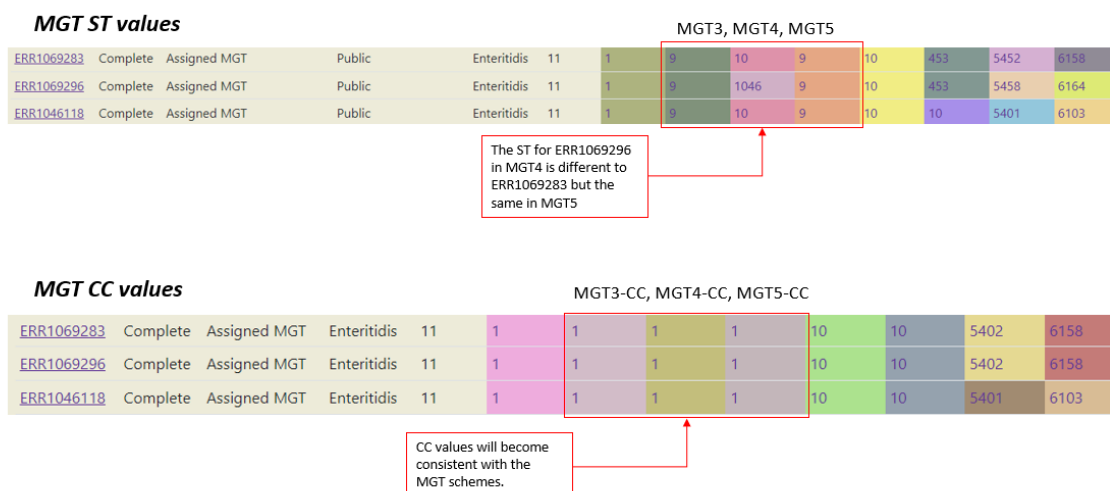


Figure 6.4: Diagram to reflect how clonal clustering is implemented on the MGT schemes to resolve Hierarchical Inconsistencies and demonstrate what a Hierarchical inconsistency is.

These new values are called Clonal Clusters (CC). MGT-CC values are unstable, since all the values are recalculated as more isolates are added into the MGT. This makes MGT-STs the more reliable identifier to look at relationships between bacterial isolates, since no recalculations are required when new isolates are added into the database. Further work to understand how problematic the hierarchical inconsistency in each species needs to be completed and a new approach or feature would need to be added to the MGTdb platform that has more stability and accurately depicts the relationships between isolates.

Chapter 7

Conclusion

To conclude, the refactoring of MGTdb was successfully completed. The refactored codebase did not affect functionality of the original program and has improved efficiency in feature development and bug fixes. I was able to confirm that the code changes did not create new bugs through testing in the frontend. Additionally, the process to locally install MGTdb was updated to account for the refactored code, thus making this process more efficient and doable to new users. The documentation for this process, which is available in MGTdb read-the-docs and the GitHub repository MGT-local's README.md file, were updated accordingly.

The codebase for MGTdb is also more efficient for installing multiple species MGTs in the same platform, thus making MGTdb more accessible to researchers. Broadly, the outcomes of the project enabled further research and application of MGT based genomic technologies to microbial science and pathogens that affects health.

References

Achtman, M., Zhou, Z., Charlesworth, J. & Baxter, L., 2022. EnteroBase: hierarchical clustering of 100 000s of bacterial genomes into species/subspecies and populations. *Philosophical Transactions of the Royal Society (Biological Sciences)*, 377(1861).

Django , 2023. *Migrations*. [Online]

Available at: <https://docs.djangoproject.com/en/4.2/topics/migrations/>
[Accessed 3 September 2023].

Django, 2023. *Models*. [Online]

Available at: <https://docs.djangoproject.com/en/4.1/topics/db/models/>
[Accessed 3 March 2023].

Django, 2023. *URL Dispatcher*. [Online]

Available at: <https://docs.djangoproject.com/en/4.1/topics/http/urls/>

Django, 2023. *Writing and running tests*. [Online]

Available at: <https://docs.djangoproject.com/en/4.2/topics/testing/overview/>
[Accessed 15 March 2023].

Docker docs, 2023. *Docker Overview*. [Online]

Available at: <https://docs.docker.com/get-started/overview/>
[Accessed 20 September 2023].

Docker docs, 2023. *Overview of the Get Started guide*. [Online]

Available at: <https://docs.docker.com/get-started/>
[Accessed 20 September 2023].

Dockerize. The Docker Hosting, 2020. *Docker python django tutorial. Dockerize a Python django App in 3 minutes..* [Online]

Available at: <https://dockerize.io/guides/python-django-guide>
[Accessed 10 October 2023].

Github Docs, 2023. *Hello World: Follow this Hello World exercise to get started with GitHub..* [Online]

Available at: <https://docs.github.com/en/get-started/quickstart/hello-world>
[Accessed 30 March 2023].

Javatpoint, 2021. *Django Templates*. [Online]

Available at: <https://www.javatpoint.com/django-template>
[Accessed 10 March 2023].

Javatpoint, 2021. *Django Views*. [Online]

Available at: <https://www.javatpoint.com/django-view>
[Accessed 10 March 2023].

Javatpoint, 2023. *Django Model*. [Online]

Available at: <https://www.javatpoint.com/django-model>
[Accessed 10 March 2023].

Kaur, S., 2022. *Visual-analytics-driven bioinformatics methods for the analysis of biomolecular data*, Sydney: UNSW.

- Kaur, S. et al., 2022. MGTdb: a web service and database for studying the global and local genomic epidemiology of bacterial pathogens. *Database: the journal of biological databases and curation*, 2022(baac094).
- Larsen, M. V. et al., 2012. Multilocus Sequence Typing of Total-Genome-Sequenced Bacteria. *Journal of Clinical Biology*, 50(4), pp. 1355-1361.
- Leinonen, R., Sugawara, H. & Shumway, M., 2011. The Sequence Read Archive. *Nucleic Acids Research*, 39(suppl_1), pp. 19-21.
- Maiden, M. C., Bygraves, J. A., Feil, E. & Spratt, B. G., 1998. Multilocus sequence typing: A portable approach to the identification of clones within populations of pathogenic microorganisms. *Proceedings of the National Academy of Sciences of the United States of America*, 95(6), pp. 3140 - 3145.
- Node.js, 2023. *About Node.js*. [Online]
Available at: <https://nodejs.org/en/about>
[Accessed 31 07 2023].
- Payne, M. et al., 2020. Multilevel genome typing: genomics-guided scalable resolution typing of microbial pathogens. *Eurosurveillance*, 25(20).
- Plekhanova, J., 2009. *Evaluating web development frameworks: Django, Ruby on Rails and CakePHP*, Philadelphia: Fox School of Business, Temple University.
- Python, 2023. *importlib - The implementation of import*. [Online]
Available at: <https://docs.python.org/3/library/importlib.html>
[Accessed 19 June 2023].
- Read The Docs, 2023. *Read the Docs: documentation simplified*. [Online]
Available at: <https://docs.readthedocs.io/en/stable/>
[Accessed 17 October 2023].
- Ruppitsch, W. et al., 2015. Defining and Evaluating a Core Genome Multilocus Sequence Typing Scheme for Whole-Genome Sequence-Based Typing of *Listeria monocytogenes*. *Journal of Clinical Biology*, 53(9), pp. 2869-2876.
- Tankeshwar, A., 2021. *Bacterial Typing Methods: Aim, Attributes, Types*. [Online]
Available at: <https://microbeonline.com/bacterial-typing-methods-aim-attributes-and-types/>
[Accessed 24 April 2023].
- Template Tags and Filters, 2022. *stringformat Filter*. [Online]
Available at: <https://www.djangotemplatetagsandfilters.com/filters/stringformat/>
[Accessed 7 July 2023].
- Tutorialspoint, 2023. *How to Use Multiple Databases with docker-compose?*. [Online]
Available at: <https://www.tutorialspoint.com/how-to-use-multiple-databases-with-docker-compose>
[Accessed 3 October 2023].
- Zhou, Z. et al., 2020. The EnteroBase user's guide, with case studies on *Salmonella* transmissions, *Yersinia pestis* phylogeny, and *Escherichia* core genomic diversity. *Genome Research*, 2020(30), pp. 138-152.
- Zhou, Z., Charlesworth, J. & Achtman, M., 2021. HierCC: a multi-level clustering scheme for population assignments based on core genome MLST. *Bioinformatics*, 37(20), pp. 3645-3646.