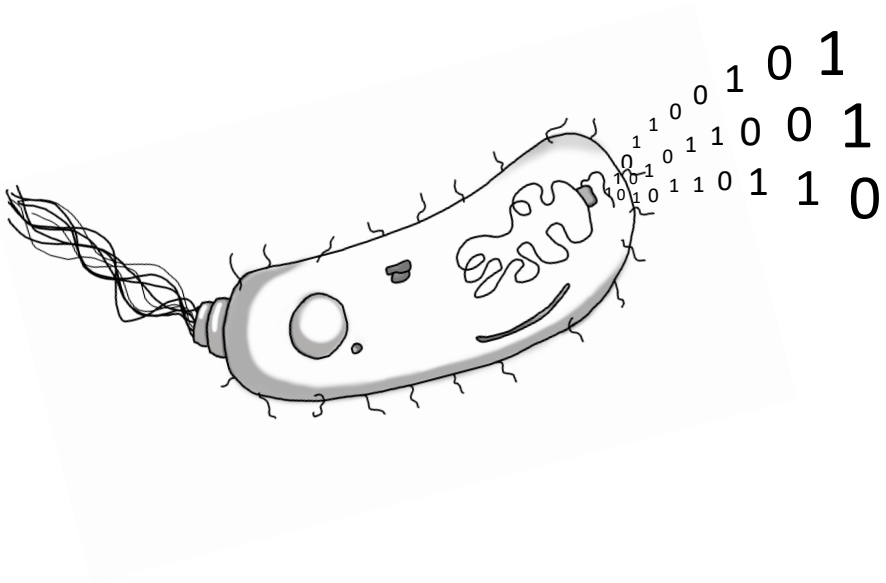
From annotation to bacterial data models

Dissertation

for the award of the degree
"Doctor rerum naturalium" (Dr.rer.nat.)



of the Georg-August-Universität Göttingen
within the doctoral program "Genome Science"
of the International Max Planck Institute Research School (IMPRS)
and Georg-August-University School of Science (GAUSS)

submitted by

Tiago Godinho de Ornelas Pedreira
from Oeiras, Portugal

Göttingen 2022

Thesis Committee

Prof. Dr. Jörg Stülke (Supervisor and 1st Reviewer)

Institute of Microbiology and Genetics, Department of General Microbiology, University of Göttingen

Prof. Dr. Burkhard Morgenstern (2nd Reviewer)

Institute for Microbiology and Genetics, Department of Bioinformatics, University of Göttingen

Dr. Johannes Söding

Max Planck Institute for Biophysical Chemistry, Computational Biology

Further Members of the Examination Board

Prof. Dr. Jan de Vries

Institute for Microbiology and Genetics, Department of Applied Bioinformatics, University of Göttingen

Prof. Dr. Christoph Bleidorn

Johann-Friedrich-Blumenbach Institute for Zoology & Anthropology, Department of Animal Evolution and Biodiversity, University of Göttingen

Prof. Dr. Stefan Klumpp

Theoretical Biophysics Group, Institute for Dynamic Complex Systems, University of Göttingen

Date of oral examination: 10.06.2022

Statement of Authorship

I hereby declare that the doctoral thesis entitled “From annotation to bacterial models” has been written independently and with no other sources and aids than quoted.

Tiago Godinho de Ornelas Pedreira

“I have not failed. I've just found 10,000 ways that won't work”

- Thomas Edison -

Acknowledgements

Embarking in a new adventure, in a country with a different culture and language can be rather daunting. I am nothing but grateful for this experience and I have learned more than what I could have ever imagined during these past three years. During this time, I counted with the great support of wonderful people.

First, I would like to thank my supervisor, Jörg, for giving me the opportunity to work in a great project, for always making it easy to work with you and for always trusting in my ability to take this and every project to the next level. I would like to thank the members of my advisory committee, Johannes and Burkhard, for always giving such important input in my project and for always putting me at ease for reaching out. I would also like to thank my students, especially Hannes and Christoph for working with me in projects that resulted in an amazing outcome. I learned a lot from you two and I am very grateful for that.

To AG Stülke and AG Rismondo also goes a big thank you, for always making the lab life much easier and bearable, especially when under such heavy restrictions during the corona lockdown. Specially, a big thanks to Neil, Lisa and Dennis for the great time together. Thank you for the small vent sessions over lunch or coffee breaks and funny moments we shared. I had a blast with you. To my PhD batch friends, Vini, Metin, Sarah, Alex, Sophie and Malte, thank you very much for all the moments we shared and the fun time we spent together. Vini, thank you for all the great talks and friendship.

A very special thank you to Larissa, from the moments in the office to now, you have taught me a lot and you truly inspire me as a person and as a scientist. Thank you for your patience, support and companionship. You made everything much easier.

To my friends Tides and Inês, a big thank you for all the moments we could share together in this adventure. Despite the distance, you were always present in my life and you've always supported me like the true friends you are. To Renato also a big thank you for the company and the good times we shared in Göttingen. Finally, to my family, a huge thank you for all the unconditional support and love you've given me. To my mom, Cristina, thank you for always being so supportive and caring, to my dad, Paulo, thank you for keeping me motivated and with my eyes on my goal, to my brother, Diogo, thank you for everything you've done for me and for always being there for me, and to my grandmother Ilda, for the grandmother's love no one else can provide but you. A vocês todos, um muito obrigado por fazerem esta jornada um pouco mais fácil.

Thank you all, for being part of this journey and for helping me achieve my goal.

List of publications

Publications within project scope:

Pedreira T., Elfmann C., and Stülke J. (2022). The current state of *Subti*Wiki, the database for the model organism *Bacillus subtilis*. *Nucleic Acids Research*, **50**: D875–D882.

Pedreira T., Elfmann C., Singh N., and Stülke J. (2022). *Syn*Wiki: Functional annotation of the first artificial organism *Mycoplasma mycoides* JCVI-Syn3A. *Protein Science*, **31**: 54-62.

Elfmann C.*, Zhu B.*, **Pedreira T.***, Hoßbach B., Senar M.L., Serrano L., and Stülke J. (2023) *Myco*Wiki: Functional annotation of the minimal model organism *Mycoplasma pneumoniae*. *Frontiers in Microbiology*. (submitted) (* - joint 1st authorship)

Michalik S., Reder A., Richts B., Faßhauer P., Mäder U., **Pedreira T.**, Poehlein A., van Heel A.J., van Tilburg A.Y., Altenbuchner J., Klewing A., Reuß D.R., Daniel R., Commichau F.M., Kuipers O.P., Hamoen L.W., Völker U., and Stülke J. (2021). The *Bacillus subtilis* Minimal Genome Compendium. *ACS Synth. Biol*., **10**: 2767–2771.

Krüger L., Herzberg C., Rath H., **Pedreira T.**, Ischebeck T., Poehlein A., Gundlach J., Daniel R., Völker U., Mäder U., and Stülke J. (2021) Essentiality of c-di-AMP in *Bacillus subtillis*: Bypassing mutations converge in potassium and glutamate homeostasis. *PloS Genetics*, **17**: e1009092.

Publications outside project scope:

Monteiro P.T.*, **Pedreira T.***, Galocha M., Teixeira M.C., and Chaouiya C. (2020) Assessing regulatory features of the current transcriptional network of Saccharomyces cerevisiae. *Scientific Reports*. **10**: 17744. (* - joint 1st authorship)

Table of Contents

Chapter 1 – Abstract

Science and technological advancements walk side-by-side and with the recent emergence of novel high throughput techniques, the necessity to have specialized data structures to host and represent the complex and high variety of information is evident. Biological databases address this major constraint and in our research group there is the focus to create these platforms to support the scientific community. Among many, *Subti*Wiki is seen in the community as the golden standard of biological databases for the model organism *Bacillus subtilis*. This platform has seen its data increase in size and quality, with highly curated information and more features to represent it. With a growing viewership, *Subti*Wiki consolidates its position among scientists by providing with novel ways to identify potential protein homologs among relatives and by integrating the popular Cluster of Ortholog Genes database. Recently, *Syn*Wiki, a biological database that shares the same framework as *Subti*Wiki was created and built to integrate data of the new synthetic organism with a minimal genome, JCVI-syn3A. Regardless of the amount of information available for both organisms, here it was shown that using the same framework is possible to expand beyond a single organism's data structure and use it for multiple organisms. Furthermore, the current state of development of this framework was evaluated, assessing its limitations in maintainability and present a novel framework that will serve as the future of all platforms created in by the research group. This framework, *Core*Wiki, was created using Flask, a minimal Python framework, that allows a modular development. Finally, the current database schema was evaluated and introduced a refreshing new one that is able to establish more robust and better relationships between the biological elements. Here, a solid contribution to all scientific fields was shown, by providing with a framework ready to integrate information from multiple levels and different organisms. Its aim is to not only organise, but to integrate the data so that every scientist accessing such platforms is able to postulate new hypotheses and take their research to new heights.

Chapter 2 – Introduction

2.1. Biological Databases

Over the past recent years high throughput methods have facilitated the exponential growth of biological knowledge. However, coupling the higher computational power with the increasing availability of high throughput techniques has raised the awareness towards data organisation and availability. More specifically, not only considering the sheer amount of data, but also its complexity rises as a major challenge in the current scientific development (Agarwala et al., 2016; Alkan et al., 2011; Loman et al., 2012; Manzoni et al., 2018; Mardis, 2017; Reuter et al., 2015). Data from next generation sequencing (NGS), proteomics, transcriptomics, metabolomics are the main contributors for this explosion of data (Loman et al., 2012; Mardis, 2017). However, and more recently, the computational power allowed disciplines such as interactomics to expand into further prediction of potential novel interactions. Interestingly, not only the classical protein-protein interactions (PPi) are predicted, but also interactions with other smaller molecules, such as metabolites and RNA (Corley et al., 2020; Gerovac et al., 2021; Link et al., 2013; O'Reilly et al., 2020). This suggests that the high variety of data demands systems that possess the flexibility to handle different types of information. These systems are called biological databases and try to answer the utmost priority via documenting and representing the data. These platforms must be in power of a solid and reliable storage system that will enable an updatable structure to provide users with fast access to data from multiple sources, which is crucial in the constant flux of new data. Finally, an intuitive interface counts as a valuable piece to give every user the possibility to have a graphical view of the integration, interpretation and contextualization of data, otherwise impossible to obtain. Indeed, biological databases play a fundamental role in connecting past discoveries to the very state-of-the-art knowledge, keeping complex information coherent, integrated and easy to access (Baxevanis & Bateman, 2015; Caswell et al., 2019).

Biological databases try to address a fundamental constraint of the growing biological data, i.e., it has limited meaning when no background or context is presented. Context is more difficult to provide when considering the immensity of data available, and so databases can opt to focus on specific aspects of research. For this, platforms can store and represent data based on the type of data they wish to include. The degree of complexity seen in databases can range from what one might consider simple to more complex data. Among these, there are databases dedicated to different natures of data, for example as part of the International Nucleotide Sequence Database Collaboration, it is possible to find the cooperative efforts of the nucleotide databases DNA DataBank of Japan (DDBJ)

(Fukuda et al., 2021), GenBank (Benson et al., 2013) and The European Archive (ENA-EMBL) (Kanz et al., 2005). Similarly, there are also databases dedicated for accessing data on proteins, such as Protein Data Bank (PDB) for protein structures (Berman et al., 2000) and UniProt for a comprehensive overview of protein sequence and annotation (Bateman et al., 2021). On a different level of information, it is also possible to find databases dedicated to data of higher complexity. Information regarding protein-protein interaction can be accessed in BioGRID (Oughtred et al., 2021) or STRING (Szklarczyk et al., 2019), while some other platforms focus their efforts on metabolic pathways, Kyoto Encyclopedia of Genes and Genomes (KEGG) (Kanehisa et al., 2021), or purely on regulatory elements, YEASTRACT (Monteiro et al., 2020). Additionally, most of these platforms provide services that allow the user to run short analysis with their element of interest. An example of this is either performing local alignments or pattern search between biological elements of interest.

Although most of these databases are focusing on single or few natures of data, some platforms are classified as metadatabases as they are databases of databases. In similarity to other popular search engines, these platforms allow each user to have full access to multiple layers and natures of information with little to no effort. An example of this type of database is the National Center for Biotechnology Information (NCBI), where the user has the possibility to make full use of multiple and different databases simultaneously. Metadatabases generally also give access to analysis tools within their structure to run local data pipelines, for example BLAST, which is embedded in NCBI (Agarwala et al., 2016; Altschul et al., 1990). This principle of “all-in-one” is greatly appreciated among the scientific community and serves as inspiration for many other databases (see more in section 2.2).

2.2. Model organisms and model organism databases

So far, we have discussed that databases are a major part of every scientist’s research, with the possibility to expand the research beyond initial limitations and postulate new theories and hypotheses. This is remarkably important for databases that instead of focusing on the nature of data, e.g., proteins, focus on a whole organism. These structures are called model organism databases (MOD) and are mostly dedicated to providing curated biological information for a specific organism, a model organism. Although other databases have undeniable utility, MODs usually are more present in the life of scientists as research groups tend to focus on biological questions applied to the specific model organism (Baxevanis & Bateman, 2015; Bond et al., 2013; O’Connor et al., 2008; Oliver et al., 2016).

Model organisms acquired this classification as they are extensively studied and used as study models for a biological question. In fact, most knowledge acquired regarding essential cell processes

has been characterized in model organisms (Fields & Johnston, 2005; Oliver et al., 2016). Understanding processes such as cell division, metabolic pathways and other conserved physiological properties allows researchers to shed light into other similar organisms. Since these organisms are extensively studied, they are also commonly referred to as the representative of a wider taxonomic group, extending most of their properties and characteristics to close species (Fields & Johnston, 2005; Oliver et al., 2016).

As a natural consequence of this intensive research, MODs play an important role to store critical information of these organisms. These platforms are usually built on a fast-pacing environment, which allows the quick integration and representation of novel data across different levels and from different sources (Bond et al., 2013; O'Connor et al., 2008; Oliver et al., 2016). Although model organisms are vastly studied, depending on each organism, the amount and type of data will differ accordingly. Evidently, the amount of data stored in MOD's relies on the ability of research groups to generate data, while the team behind MOD's are responsible for ensuring the quality of curation for the specific organism of interest. This translates to an equally extensive work from the MOD development team, in which more than gathering the data, it is responsible to organise it and frame it in its proper context by combining it with current knowledge. Indeed, the efforts towards the construction of a successful MOD lie between the balance of data generation and the quality of its curation, thus providing an organised view over the now structured, multi-nature and multi-source information. Only then all conditions are satisfied to fully support the research of millions of scientists around the World, providing with the necessary tools to push further the boundaries of knowledge as we know it (Bond et al., 2013; Fields & Johnston, 2005; O'Connor et al., 2008; Oliver et al., 2016).

Currently, it is possible to find information for virtually all model organisms in dedicated databases. The type of data can, however, differ from database to database as these can have a different nature of data focus. An example of this is the MOD for the well-known budding yeast *Saccharomyces cerevisiae*, Saccharomyces Genome Database (SGD) (Cherry et al., 2012), which serves as an extensive compendium of most data for this organism, and the database YEASTRACT (Monteiro et al., 2020) that focuses mainly on transcriptional regulators of the same organism. Another noteworthy MOD is EcoCyc, dedicated to the model Gram-negative bacterium, *Escherichia coli*, which continues to grow in popularity (Karp et al., 2018; Keseler et al., 2021). The Gram-positive bacterium *Bacillus subtillis,* a widely researched model organism, also has its own spotlight counting with multiple MODs (to be discussed further ahead). Moving out of the microscopic world, other organisms also have their own dedicated databases, for example the information of the fruit fly *Drosophila melanogaster* can be found in FlyBase (Larkin et al., 2021), the documented data for the mouse *Mus musculus*, widely used in experiments, can be accessed in Mouse Genome Informatics (Bult et al.,

2019), and the model plant *Arabidopsis thaliana* relies on The Arabidopsis Information Resource to keep track and document all available information for this organism (Berardini et al., 2015). In this work, the focus will be mainly on *B. subtilis* and its database *Subti*Wiki, as well as the ability to scale these structures to different organisms.

2.3. *B. subtilis* and *Subti*Wiki

As a natural consequence of its intensive investigation, *B. subtilis* is considered the defining organism among Firmicutes and the model Gram-positive bacterium (Errington & van der Aa, 2020; Kovács, 2019). It is a fast-growing aerobic organism with important traits that back up its popularity. Despite not being pathogenic, it can produce heat-resistance spores, is able to create biofilms and is a close relative to many pathogenic species. *B. subtilis* is seen as a workhorse in the biotechnology industry due to its genetic competence, easy laboratorial manipulation, fermentation properties and the ability to secrete compounds, such as vitamins (Arnaouteli et al., 2021; Errington & van der Aa, 2020; Kovács, 2019; Zweers et al., 2008). Moreover, *B. subtilis* has also been used as a model organism in the efforts to understand the basic principles of life in the process of creating minimal organisms (Michalik et al., 2021; Reuß et al., 2016, 2017).

As a result of the advancements in knowledge for this organism, it has seen numerous databases created with the purpose of keeping data organized and up to date (Table 2.1). Most databases created are vastly dedicated to a specific aspect of data, with few exceptions. For example, the Database of Transcriptional Regulation in *B. subtilis* (DBTBS) focuses on the regulatory elements in this organism (Ishii et al., 2001; Sierro et al., 2008). *Subti*List and *B. subtilis* ORF Database (BSORF), two of the first relational databases dedicated to *B. subtilis*, were created in response to the, at the time, newly sequenced *B. subtilis* genome. They aimed at documenting all information associated with the genetics of this microorganism as well as introducing some tools for analysis, such as pattern matching (Moszer et al., 1995, 2002; Ogiwarea et al., 1996). However, these platforms have been outdated and the alternative was a database, BsubCyc, that although up to date, is sitting behind a monthly membership (Karp et al., 2018). Thus, there was the need for the emergence of a novel open platform for all community to have access to curated data in the most novel and intuitive way possible. With this objective in mind, *Subti*Wiki was first born in 2009 (Flórez et al., 2009) and aimed at filling the necessity among the *B. subtilis* community to have a data structure with state-of-the-art data and free access.

Table 2.1 – Available *B. subtilis* databases. Information regarding the focus of data and last time of update displayed accordingly.

Database	Focus of information	Comment
DBTBS	Regulatory information	Last updated in 2008
*Subti*List	Functional annotation	Last updated in 2004
B. subtilis ORF	Functional annotation	Last updated in 2006
BsubCyc	Functional genome annotation	Membership since 2017
*Subti*Wiki	Functional annotation	Started in 2008

As an initial implementation, *Subti*Wiki relied on a MediaWiki engine to generate views (pages) to represent the available data. Since the deployment of *Subti*Wiki v1, the gene has been the central element of the database, which allows to establish relationships with every interacting biological or chemical element (Flórez et al., 2009; Mäder et al., 2012; Michna et al., 2014, 2016; Zhu & Stülke, 2018). For example, the *dnaA* gene represents the Entity **dnaA**, and has a relationship with the protein **DnaA**. Because each gene is to be treated as a main Entity, gene names are used as identifiers and their information must be curated throughout to avoid misuse among the scientific community. To simply put, in the process of organising the data, *Subti*Wiki ensures that all **dnaA** related information, e.g., synonyms, locus tag, etc., is referenced properly. This guarantees that no information is lost or is redundant in the process of data collection and storage.

By revolving data around the entity **Gene**, gene pages display fundamental information such as synonyms, essentiality, gene product, function, molecular weight, gene and protein length, and genomic context. In the first iteration of *Subti*Wiki, an integrated metabolic and regulatory pathway visualizer was introduced (Flórez et al., 2009), and later provided a protein-protein interaction graphical representation (Mäder et al., 2012). Along years, more data was included in the following versions of *Subti*Wiki, enriching the contents of a database that was already on the spotlight among the community. The implementation of other database unique identifiers, integration of a comprehensive collection of transcription factors and regulons, creation of tailored functional categories that govern *B. subtilis* (Michna et al., 2014), visualization of expression data under different conditions (Michna et al., 2016), a fully-fledged Genome Browser and Biological Network visualizer with the option to overlay expression data (Zhu & Stülke, 2018), are just a few of the major improvements this platform has seen over the past years. Impressively, all iterations of *Subti*Wiki count with the manual curation of the data, containing over 6000 protein- and RNA-coding genes, more than

1790 regulatory information entries, 49 metabolic pathways and over 6000 publications annotated among all genes (Zhu & Stülke, 2018).

2.4. JCVI-syn3.0 - the minimal genome cell

As minimal genomes gained popularity among the *B. subtilis* community, the attention towards synthetic biology has shifted (Reuß et al., 2016, 2017). This field takes inspiration in nature to fully understand what lies behind the essential requirements for life. Drawing knowledge from the very essentiality of genes, this emerging field tries to understand the principles of life. Creating artificial cells envisions to shed light on which are the responsible players in the basics of sustaining life, with the immense potential to revolutionize the fields of biotechnology and medicine (Cameron et al., 2014; Garner, 2021; Khalil & Collins, 2010).

Synthetic biology allows to engineer cells with desirable characteristics and thus, it is necessary to trim down genomes to the very core of their essentiality. For this, two possible methodologies can be used: the top-down approach, where essential genes and functions are identified in order to successfully predict which genes are fundamental for life and consists of consecutive genome reductions until a minimal organism has been achieved. Taking full advantage of its status as model organism and as mentioned before, *B. subtilis* has been target of successful studies to reduce its genome using this approach (Reuß et al., 2016, 2017). The counterpart of this methodology, the bottom-up approach, focuses on the *in vitro* synthesis of minimal genomes, which are then assembled and transplanted into a cell, giving full meaning to engineering a custom-made genome (Schwille, 2011). While the first approach is better used when there is extensive knowledge on the organism, the latter one seems to be the choice when approaching a novel organism, as it was the choice to build the novel synthetic microorganism *Mycoplasma mycoides* JCVI-syn3.0 (Hutchison et al., 2016).

It is possible to argue that being able to reduce a genome this much and still have a viable cell at the end, would be the consequence of the established knowledge over the essential genes of such organism. However, this is far from the truth, as among its 452 protein-coding genes there is a big portion of unknown proteins (Hutchison et al., 2016). Shedding light into the vast unknown of a synthetic organism has been seen to be a challenge. From the conceptualization to the living cell, little is known regarding the true essential functions of genes, which only now begins to be unveiled with the emergence of novel studies (Breuer et al., 2019).

2.5. Expanding *Subti*Wiki framework to other organisms

The emergence of novel organisms such as JCVI-syn3.0 highlights the importance to have tailored structures, ready to use and update, to host and represent data. There are, as expected, some considerations to have when designing such concept. A framework can be tailored to a specific purpose and excel at it, or it can be optimized so serve a vast number of purposes. Although expanding from *B. subtilis* to different prokaryotes models is possible, some constraints are to be taken in consideration. While for *B. subtilis* data is available in different formats and amounts, to the extent that it represents a bottleneck, for some not so well-studied organisms, such as the JCVI-syn3.0, the lack of data is the greatest limitation. Indeed, all biological database frameworks should contemplate with effective methods to store and represent data from any source and respond to the growing amount of information (Baxevanis & Bateman, 2015). *Subti*Wiki has responded to this in a successful manner, continuing to add state-of-the-art information, generating novel ways of representing complex data and continuing to grow in popularity, counting with millions of requests yearly (Flórez et al., 2009; Mäder et al., 2012; Michna et al., 2014, 2016; Zhu & Stülke, 2018). It is clear that *Subti*Wiki data models are well established, and by taking the principles of its models, it is possible to try to apply them to other organisms. This is particularly valid for prokaryotes due to their genomic similarities.

Notably, this data structure has shown record on its usefulness and application in modulating other bacterial data models in a similar capacity. An example of this is the database for the microorganism *Mycoplasma pneumoniae* in *Myco*Wiki (www.mycowiki.uni-goettingen.de) and *Listi*Wiki (www.listiwiki.uni-goettingen) for *Listeria monocytogenes* (unpublished data). Although the integration of the current scarce available documentation is successful, the great test for *Subti*Wiki's framework relies on allowing to build a database from scratch when taking close to no data. However, to achieve such feature it is important to take into full consideration all limitations of the current structure and develop an approach accordingly.

2.6. Limitations of a data model framework

As any other technological application, data structures are expected to be on par with, in this case, the data generation and evolution of technology. To be able to respond quickly to the pacing of these two independent variables, there are some limitations that need to be understood.

With no surprise, the first limitation is tied to the data itself. While it is undeniable that *Subti*Wiki takes full advantage of its custom-made framework to respond to the increasing amount of data that is generated yearly, there are some struggles when implementing data from different natures. As stressed before, not only the nature and novelty of data pose as a major challenge, but

also its sheer amount that raises concerns (Baxevanis & Bateman, 2015; Mardis, 2017). Here, the issue is instead bound to the selection of which data to represent, for the simple reason that not every emerging data can be deemed worthy of inclusion in a database. Indeed, the tsunami of information that is generated needs to be investigated and critical thinking becomes mandatory to choose which data should be included and, thus, represented in a data model. To tackle this, the designated function of curators is imperative in the maintenance of balance between the amount of data that is generated and the data that is meant for representation (Caswell et al., 2019).

The second limitation, although tied to the data, is also bound to the technological advancements, because with new technology, novel ways of representing old and new data may arise. The development of new features of a database is intrinsically connected with the available technology and data. For example, the high complexity of protein interaction data can be represented using an undirected network, where edges would represent the interaction and nodes the proteins (Biggs et al., 1976). Understanding what lies behind the data, how to translate a biological interaction to graph's theory, is the key for a successful representation. Protein-protein interaction maps have been used for quite some time already, however when considering how to introduce novel elements to the same representation might quickly become a challenge. While this may be a simple example, it serves the purpose to stress that fully understanding the data and how to connect it with the available technology to represent it, is a requirement for every framework to enable users to reach new depths in an ocean of information. Limitations tied to all aspects of data are not necessarily something a framework can fix, due to its ever-changing nature. However, it is important to retain that, when building a framework, it should be as flexible as possible by creating robust yet plastic data models. This consideration becomes a tough challenge when specifically adapting to new organisms. This is mostly caused by the potential changes in the data structure that are fundamentally based on the organism's data nature. For example, as an effect of evolutionary pressure towards an AT rich genome, *Mycoplasma* diverged from other organisms towards an AT rich genome, instead of the CG counterpart. In practical terms, this will affect how the database will recognise the **TGA** codon: tryptophan for *Mycoplasma* and stop codon for other organisms (Inamine et al., 1990; Rocha et al., 1999; Yamao et al., 1985).

When expanding to different organisms, more than knowing the infrastructure, it is crucial to have full understanding over which data to include. By ensuring that, the developer has full control over the information and adapting data models becomes an accessible task during the process of creation and maintenance of the database.

2.7. Maintainability of a data model framework

Considering a more comprehensive view over database frameworks, there are some demands regarding the quality of the framework that need to be attended. To better understand these demands, ISO/IEC 25010:2011 Systems and software Quality Requirements and Evaluation (SQuaRE) provides general good practice guidelines that should be taken into consideration (Estdale & Georgiadou, 2018). Figure 2.1 displays the categories that govern these guidelines. Interestingly, there are two major categories, **Quality in Use** and **Product Quality**. While the first tries to address the usability of the framework by the end user, the latter contains characteristics that are of interest for the developer. In the scope of this work, there is a particular attention and dedication to the category **Maintenance**, under **Product Quality**, as it plays a crucial role addressing the data model limitations in the development of biological databases.

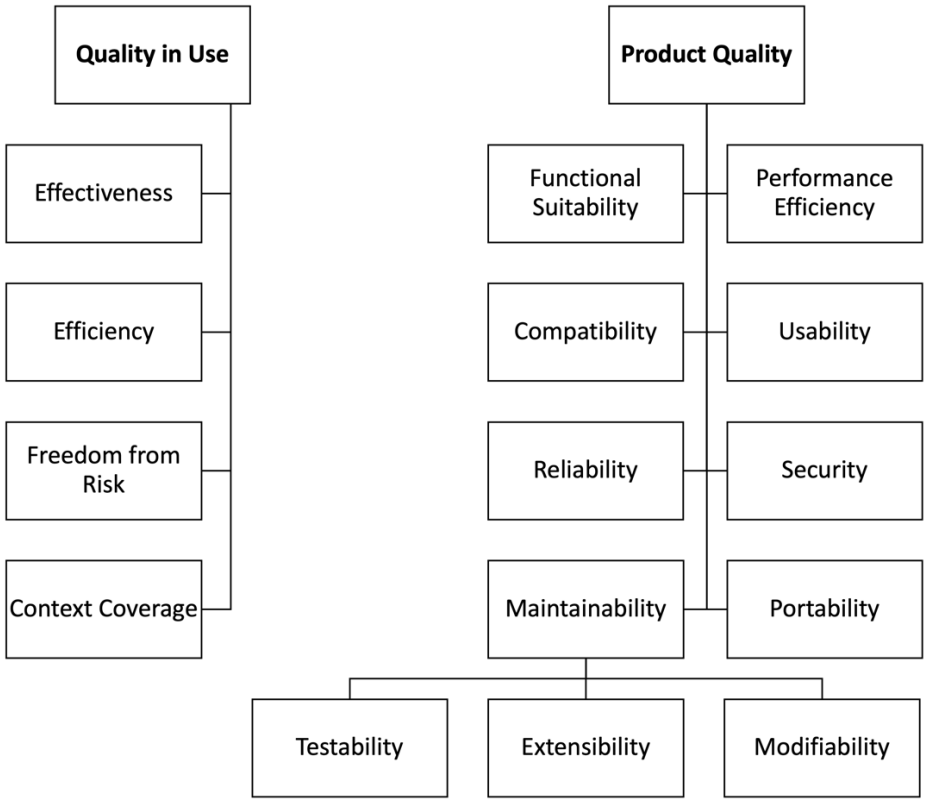


Figure 2.1 – ISO/IEC 25010:2011 Systems and software Quality Requirements and Evaluation suggested categories for an application of software (Estdale & Georgiadou, 2018). There are two major categories: Quality in Use, which targets the end user as its main objective to perform; Product Quality, which contemplates series of categories developers of software should be guided by.

For a biological database framework, maintainability is classified as the degree to which the structure can be modified by the developers or even end users. Within this category there are three more important subcategories that play a fundamental role altogether:

- **Testability**, which allows components to be tested independently and automatically
- **Extensibility**, which asserts modular extension of novel features without the need to construct from scratch
- **Modifiability**, which describes how efficiently and effectively the framework can be altered without affecting its current state

These subcategories are considered key players in the integration of emerging high throughput data, as they aim to respond to the very core issue with the data limitation in biological databases. Accordingly, it is safe to acknowledge that the flexibility of a framework can be measured by how much it needs to change in order to successfully integrate and host novel data.

2.8. Aim of the project

The major success of *Subti*Wiki as a database that serves not only the *B. subtilis* community, but the microbiology research community in general, is without a doubt undeniable. With many applications that go beyond this organism, *Subti*Wiki has established itself as the leading platform for *B. subtilis* and close relatives and keeps aiming further and beyond. With the emerging novel data from high throughput techniques, the need to build specialized handling structures plays a crucial role. This, however, comes at the expense of having to develop novel methods to integrate current and new data. Approaching this challenge with the current *Subti*Wiki structure, this project envisions to integrate novel data on multiple levels into a new version of this platform. Generating more information leads to new opportunities in the development of data analysis pipelines that aim to discover new aspects of an already extensively studied organism. Taking the current documented data and using it to run further analysis in the discovery of potential protein homologs, for example, will help consolidate the current state of *Subti*Wiki in the research community. Beyond organising recent data, there is the aim to contribute by adding novel features alongside in-group generated information.

As expected, increasing information is also observed for other and novel organisms. For example, the newly synthetic microorganism, JCVI-syn3.0, holds a minimal genome and aims to shed light into the basic principles of life, by unveiling which genes are truly essential for life to exist. Although a considerable portion of its proteins is still of unknown function, it presents itself as the perfect candidate to test the very limits of *Subti*Wiki's framework when expanding to different organisms. Thus, all conditions are now gathered to test the limits of this framework, while providing

the community of synthetic biology with an important platform that can be a hallmark in the documentation and annotation of this novel organism.

There are, however, more concerns to be addressed on a more technical and developmental side of each Wiki's framework. Although *Subti*Wiki has an excellent record of publication and user engagement, there are some challenges in the development of the current platform. One of the most important aspects of any framework for the developer is to have access to its documentation. *Subti*Wiki is built on a custom-made framework written in PHP, Hypertext Preprocessor, which is a widely used programming language for web development (Bakken et al., 2000). In contrast with other well-established and open-source public frameworks, and due to its custom origin, the current framework of *Subti*Wiki does not have the necessary documentation to support further development. However, it is important to note that being built in PHP and being a custom framework does not have a direct correlation with the challenges of its development and should have no impact in its performance and maintainability.

Regardless of being very flexible and working as intended for the current data, there are some problems when trying to develop novel features that aim at integrating novel or even already existing data in *Subti*Wiki. An example of this is when a biological element is updated, e.g., adding information on which Protein family it belongs to, the framework already supports the necessary infrastructure to provide this update. However, to implement interaction data of higher complexity, such as protein-RNA or protein-metabolite interactions, new data models and representation must be generated. Even when considering novel data that emerges for specific elements, very often there is the need for the framework to undergo some rework or adaptation to accommodate it. Importantly, the degree of modification is usually correlated to the existence of an instance of that nature of data in the database. While having to actively create more modules to the framework is to be expected, the infrastructure lacks the necessary documentation to guide through this process. As it is possible to understand, without the proper "blueprints" to navigate in the framework, it becomes nearly impossible to identify and fix issues, as well as to develop new components for different features. These issues present themselves as a big gap in the development of *Subti*Wiki and other Wikis, and more strategies are to be considered to continue providing the community with strong and up-to-date platforms.

A potential approach to address developmental hurdles is to create a new platform, based on an existing open-source framework, and make use of the full extent of its documentation to provide each user and developer a better experience. Moreover, as a consequence of increasing each Wiki's maintainability, scaling this framework to other organisms and even outside of our research group, becomes a feasible achievement. Thus, this project evaluates other frameworks that can be used in an alternative approach to build a biological database. For example, which other programming languages

might be able to support the development in a more dynamic way, providing with the necessary tools for the task and enough room for future novel features. Changing the programming language behind the framework does not affect the final vision of our structures, but rather, consistently enables the expansion of its functionalities and maintainability, while providing all necessary documentation to support its development.

Since there is the motivation to greatly change the framework underlying any Wiki project, there is also the necessity to evaluate the current database scheme. More specifically, how to convert the current database into a more modern and robust scheme with self-running and well-established maintenance routines, making it easier than ever to access and edit data. Overall, there is the will for the developer of the Wikis to be presented with a much friendlier code, a well-outlined modular structure and the proper documentation to aid in the development of novel features, with a strong and solid data model to support its data.

Chapter 3 – Development of *Subti*Wiki v4

The results described in this chapter were originally published in *Nucleic Acids Research* (DOI:10.1093/nar/gkab943):

The current state of SubtiWiki, the database for the model organism Bacillus subtilis

Tiago Pedreira[1], Christoph Elfmann[1] and Jörg Stülke[1]

[1] Department of General Microbiology, GZMB, Georg-August-University Göttingen, Germany

Author contributions

Tiago Pedreira: Conceptualization (lead); investigation (lead); methodology (lead); project administration (supporting); resources (lead); supervision (lead); validation (equal); visualization (equal); writing; original draft (equal); writing; review and editing (equal). Christoph Elfmann: Methodology (supporting); resources (supporting); visualization (supporting). Jörg Stülke: Conceptualization (equal); data curation (equal); funding acquisition (lead); project administration (lead); supervision (lead); writing; original draft (equal); writing; review and editing (equal).

3.1. Abstract

Bacillus subtilis is a Gram-positive model bacterium with extensive documented annotation. However, with the rise of high throughput techniques, the amount of complex data being generated every year has been increasing at a fast pace. Thus, having platforms ready to integrate and give a representation to these data becomes a priority. To address it, *Subti*Wiki (http://subtiwiki.uni-goettingen.de/) was created in 2008 and has been growing in data and viewership ever since. With millions of requests every year, it is the most visited *B. subtilis* database, providing scientists all over the world with curated information about its genes and proteins, as well as intricate protein-protein interactions, regulatory elements, expression data and metabolic pathways. However, there is still a large portion of annotation to be unveiled for some biological elements. Thus, to facilitate the development of new hypotheses for research, we have added a Homology section covering potential protein homologs in other organisms. Here we present the recent developments of *Subti*Wiki and give a guided tour of our database and the current state of the data for this organism.

3.2. Introduction

With the rise of high throughput techniques, the complexity and amount of information has been increasing significantly over the past years. Importantly, the new techniques not only allow the acquisition of one-dimensional data sets such as new sequences or protein catalogs, but they can generate bi- or even multidimensional data sets that include two or multiple partners such as in global protein-protein, protein-RNA, or protein-metabolite interactomes (Gerovac et al., 2021; Link et al., 2013; O'Reilly et al., 2020). A possible approach to catalogue and organize the data is by resorting to specialized databases. One type of these databases is dedicated to model organisms, which play a fundamental role in the advance of knowledge in science as they are widely used across the world (Dietrich et al., 2014; Oliver et al., 2016). The databases can widely differ in their intended audience and purpose, and accordingly, the presentation of the data varies substantially. Clearly, databases with a general audience in mind need a more intuitive structure and way of presentation as compared to databases that are designed for informaticians who are mainly interested in the technical possibilities to extract information.

The Gram-positive model bacterium *Bacillus subtilis* is one of the most intensively studied organisms. This bacterium has gained much interest due to the simple developmental program of sporulation, its genetic competence and the ease of genetic manipulation, because of the wide applications in biotechnology and agriculture, and since it is a close relative of many important pathogens such as *Staphylococcus aureus*, *Clostridoides difficile*, or *Listeria monocytogenes* (Errington

& van der Aa, 2020; Kovács, 2019). We have previously developed *Subti*Wiki, a database focused on the functional annotation of *B. subtilis* (Michna et al., 2014, 2016; Zhu & Stülke, 2018). In *Subti*Wiki, the user has access to individual gene and protein pages, complemented by *Subti*Apps, which give access to integrative metabolic pathways, regulatory networks and manually curated protein-protein interaction maps (Zhu & Stülke, 2018). As of today (September 2021), *Subti*Wiki contains data on 6,121 protein- and RNA-coding genes, 2,271 operons, 8,355 literature references and more than 2,660 documented protein interactions. Additionally, 49 metabolic pathways are displayed in an interactive map, a genomic interactive map is provided for the full extent of the *B. subtilis* genome, and gene expression pattern profiles are presented for 121 experimental conditions, which can be on the transcript or protein levels.

Despite the current advances in the research on *B. subtilis*, there is still a large portion of genes to be further annotated. This is the case for about 30% of all *B. subtilis* genes/ proteins. We try to address this issue by providing the user with a broad list of protein homologs in different organisms. To complement this, we also integrated the Cluster of Ortholog Genes Database (COG) for every available protein (Galperin et al., 2021). Thus, beyond providing the community with an up-to-date and free to use platform, we also contributed to the annotation of this model organism. Here, we report the current state of *Subti*Wiki, the integration of new data, and describe the most recent addition, the Homology Analysis.

3.3. *Subti*Wiki gene pages

Similar to previous versions, version 4 of *Subti*Wiki keeps the general design and page layout. Upon searching for a gene or protein, the respective biological element page is presented in a simple and intuitive way (Figure 3.1). Specific and factual information is promptly presented in the form of a table, such as data on locus name, and in the case of protein-encoding loci, isoelectric point, molecular weight and function. Additionally, it is possible to directly access NCBI Blast tools with either DNA or protein sequences and the user gets access to other relevant database links that also have information on the queried protein, such as KEGG, UniProt, or BsubCyc. Moreover, curated interactive information on specific data such as protein structure, protein-protein interactions, regulatory elements and functional annotation is also available. Each gene/ protein is assigned to one or more functional categories organized in a tree-like structure and to regulons. The pages then provide detailed information on the gene, the protein, and the regulation of its expression. To complement the data, biological materials available in the community, laboratories that study the gene or proteins, and publications are presented in a categorized way for easy reading and organization.

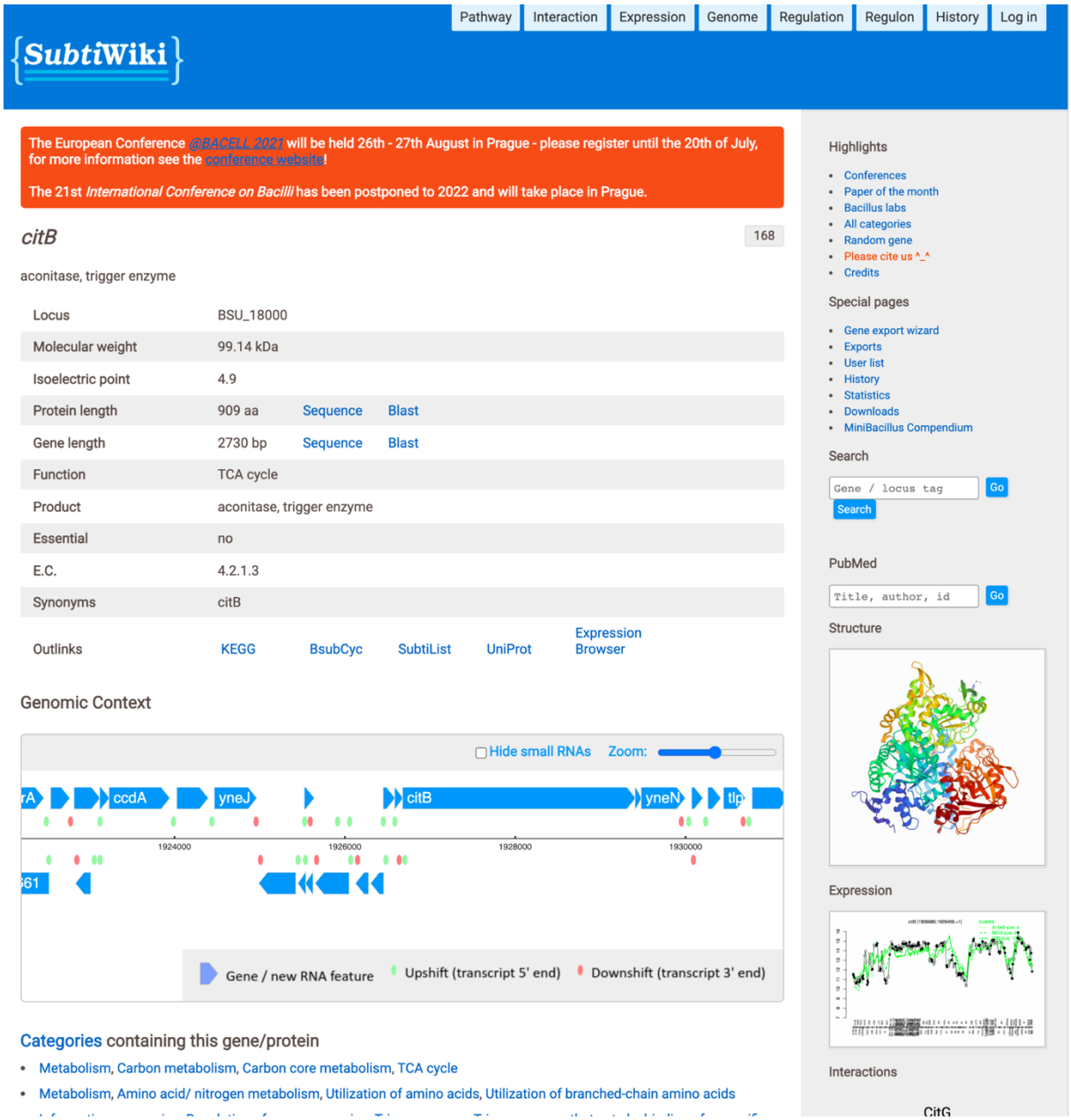


Figure 3.1 – The *citB* gene page display. All gene pages share the same layout and follow the same structure, with changes that depend on the type of available data.

3.4. *Subti*Apps

For a better representation of information, different types of data are presented under specific browsers in *Subti*Apps. The Expression Browser fully explores expression patterns under 121 experimental conditions, while the Genomic Browser is an interactive map where one can scroll through the *B. subtilis* genome in an intuitive and easy manner. In addition to genes' coordinates, their

respective DNA and protein sequences are loaded to provide a complete view over the genome (Figure 3.2).

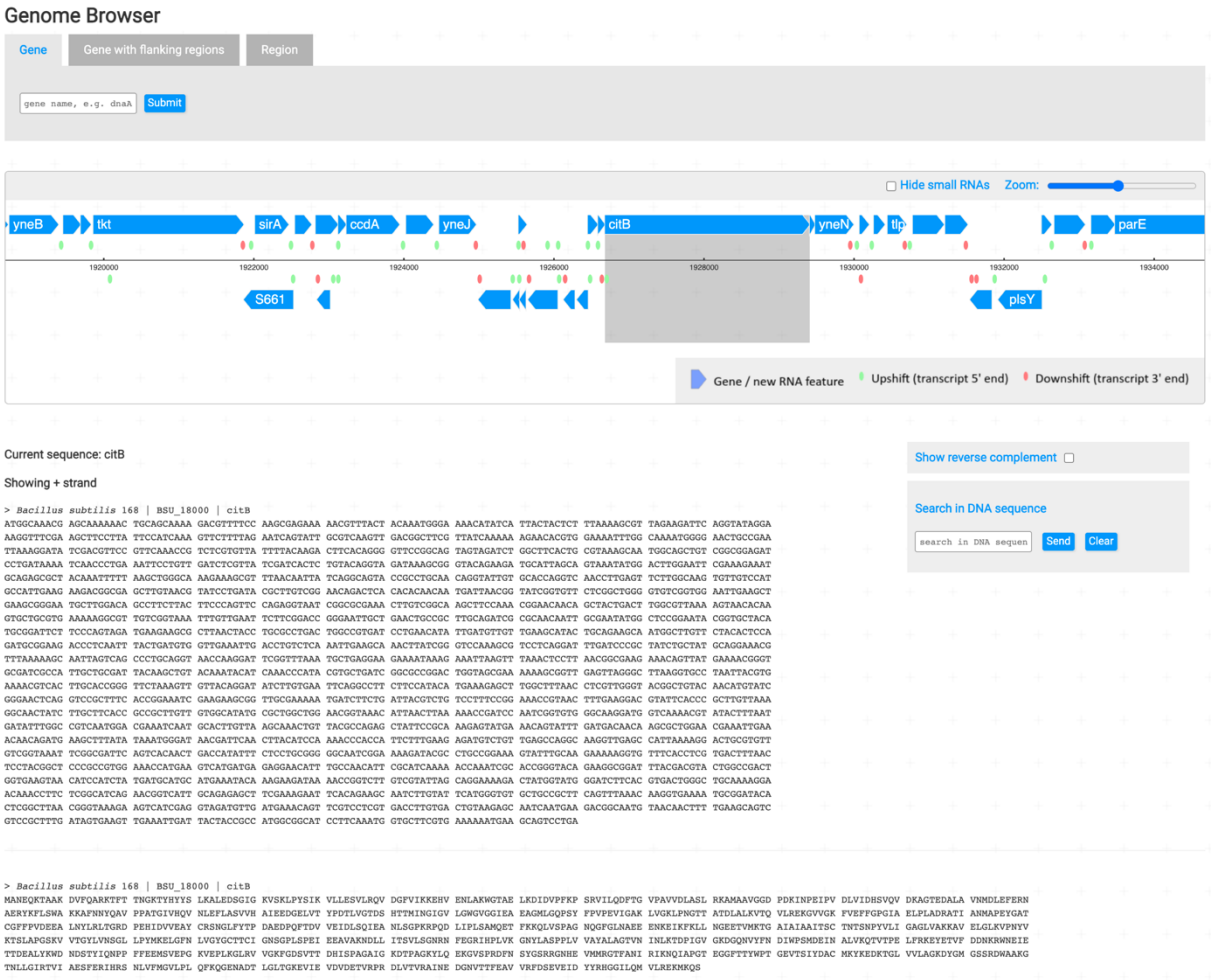


Figure 3.2 – Genome browser focusing on the *citB* gene. Tabs at the top allow to search for specific genes, include flanking regions, or simply to query by region. The main frame includes an interactive scrollable genome, where all elements are clickable for a redirect to the specific gene page. At the bottom, both protein and DNA sequences are shown (on the left) with the possibility to show the reverse complement or to search for specific patterns within a sequence (on the right).

One of the highlights of the previous version was the biological network visualizer for interactions and regulation (Zhu & Stülke, 2018). Currently, we have expanded the data to more than 5,950 and 2,660 experimentally validated regulatory and protein-protein interactions, respectively. Here, we offer a clear representation of the documented physical interactions with the possibility to expand the neighborhood (Figure 3.3A). Focusing on the regulatory data, regulatory networks can be interrogated for 2,271 documented operons. There are three different types of annotation depending on the

mechanism of regulation: green for positive regulation (or activation), red for negative regulation (or inhibition) and gray for sigma factor interactions (Figure 3.3B). Similar to the interaction browser, the regulation browser also allows to expand the view to the neighborhood of the target gene, giving the user a view over the full known regulatory network of the gene of interest, but also of its neighbors, even allowing the visualization of regulatory sub-networks (Figure 3.3C). The gravity model (Fruchterman & Reingold, 1991) in which nodes (genes/ proteins) are considered as mass points and edges (physical or regulatory interactions) as springs was preserved for both browsers and the user can find a clear and intuitive integration of the expression data within networks (Figure 3.3A). Notably, all references of such data can be found at the bottom of each respective gene page in the References section.

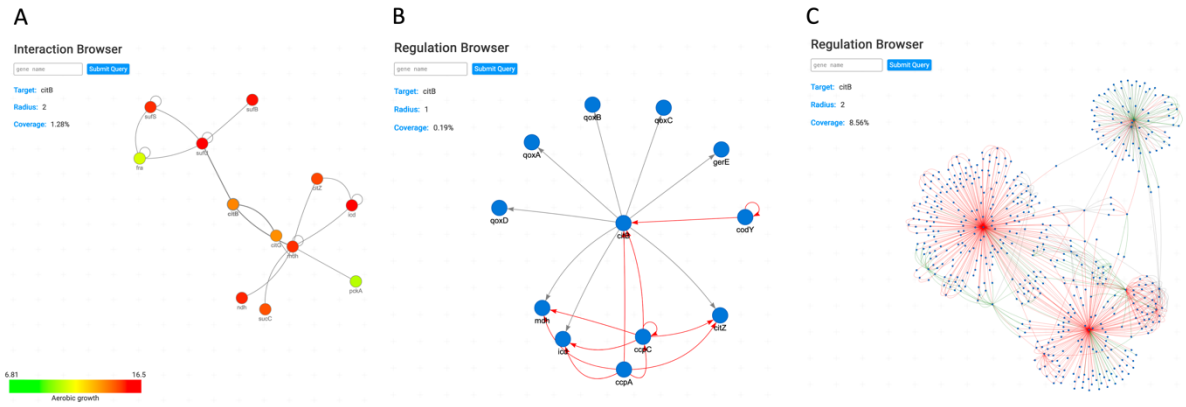


Figure 3.3 – The Interaction and Regulation browsers. (A) Interaction browser for the *citB* gene. Nodes represent proteins and edges represent physical interactions, commonly referred to as protein-protein interactions. Using the settings, transcriptomic and proteomic data can be integrated using a color code (this feature is also available for the Regulation browser). Upon selection of an experimental condition to be visualized, each node will be masked with a color depending on the expression value of the gene in the selected dataset. Green represents lower values of expression and red represents higher values of expression. (B) Regulation browser for the *citB* gene. Nodes represent genes and edges represent regulatory interactions. The interaction type is represented with a color scheme: red for repression, green for activation, gray for sigma factor. (C) Regulation browser for the *citB* gene with an increased radius of distance of 2. This setting allows to expand all regulations from the initial representation (Fig. 3B) with the distance of 2 elements (thus including the regulatory interactions of the primary partners that are shown in (B)), creating a more complex view over the regulatory network of *citB*.

3.5. Potein homology integration

According to the *Subti*Wiki data collection, there are 1,819 biological elements with unknown function or lacking annotation, which represents about 30% of all elements. To mitigate this lack of annotation, we decided to integrate information on potential protein homologies for all *B. subtilis* proteins. For this, we created a proteome library of 16 relevant bacteria, some close relatives of *B. subtilis* as well as important representatives of other bacterial groups such as the proteobacteria or cyanobacteria (Letunic & Bork, 2021) (Figure 3.4A) by extracting the corresponding sequences from UniProt (Bateman et al., 2021). To assess potential homologs, we developed a pipeline using the well-known sequence similarity search tool FASTA (Madeira et al., 2019). The pipeline is composed of two rounds of alignments. The first one compares every protein of *B. subtilis* with the library, and the second one compares the best hits found in the previous step with the *B. subtilis* proteome. If the resulting protein from the second alignment is the same as the initial input of the first step, then the proteins are considered bidirectional best hits (Figure 3.4B). Although performing this second round of alignments gives more confidence in the prediction of homologs, it was still necessary to consider the difference of protein sizes when running alignments. Disregarding this step may cause false positives, as small proteins can have a high identity score for multiple regions of larger proteins. To avoid this, we paid special attention to the alignment of smaller proteins with larger ones, and manually excluded cases where strong inconsistencies were found. All resulting pairs showing an E-value $\leq$0.01 with a minimum identity of 40% were considered. However, lower values of identity were also considered as long as high values of similarity were retained.

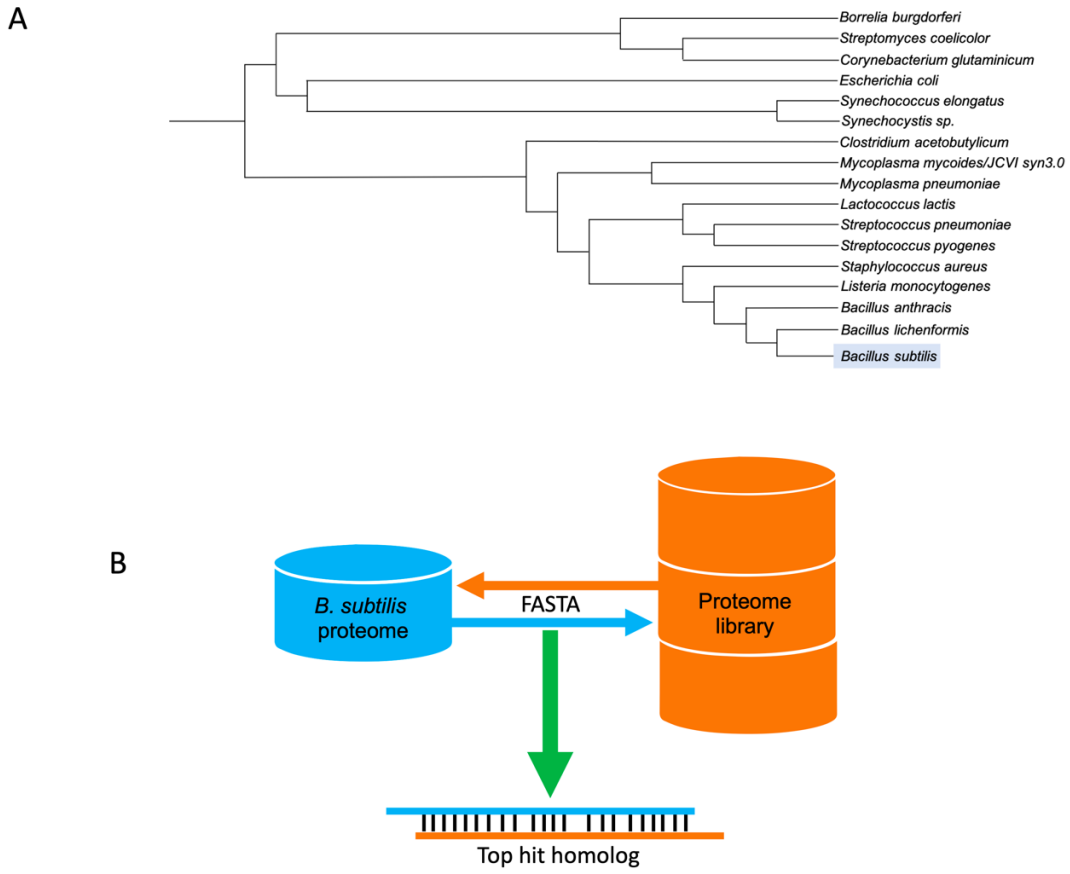


Figure 3.4 – Homology analysis. (A) Phylogenetic tree of all organisms chosen for the protein homology analysis. (B) Bidirectional FASTA alignment pipeline representation.

Currently, *Subti*Wiki provides protein homology results for every protein, with the exception of 254 proteins that did not meet the identity/similarity and/or E-value cutoff criteria for any of the representative relatives. The results can be accessed on the gene page and are presented in an easy-to-read table (Figure 3.5). For each queried *B. subtilis* protein, there is a list of organisms and their respective best hit protein homologs. Each of these proteins is linked to its respective UniProt page and the metrics for identity, similarity, and bidirectional best hit are shown. If no protein homolog for a certain organism was found, then this is stated (Figure 3.5).

Potential protein homologs

Description

The following list presents the potential protein homologs for CitB using FASTA alignment tool. For all top hits, a bidirectional analysis was performed and those showing the same pair of proteins are considered to be potential orthologs. The column ***Best hit bidirectional*** informs if CitB was found to be best hit in the alignment with the protein of the respective organism.

CitB

Organism	Protein name	Identity	Similarity	Bidirectional best hit
Bacillus licheniformis	CitB	90.9%	97.7%	Yes
Bacillus anthracis	AcnA	78.7%	92.5%	Yes
Listeria monocytogenes	CitB	73.3%	91.5%	Yes
Staphylococcus aureus	SAOUHSC_01347	72.0%	90.4%	Yes
Lactococcus lactis	CitB	54.9%	80.1%	Yes
Streptococcus pneumoniae		No significant homolog	No significant homolog	
Streptococcus pyogenes		No significant homolog	No significant homolog	
Clostridium acetobutylicum	CitB	25.7%	48.7%	Yes
Mycoplasma pneumoniae		No significant homolog	No significant homolog	
JCVI-syn3A		No significant homolog	No significant homolog	
Corynebacterium glutamicum	Acn	50.5%	75.5%	Yes
Streptomyces coelicolor	CAC37548	55.1%	79.4%	Yes
Escherichia coli	AcnA	56.6%	80.8%	Yes
Synechocystis sp.	LeuC	27.5%	52.4%	No
Synechococcus elongatus	LeuC	27.1%	51.6%	No
Borrelia burgdorferi		No significant homolog	No significant homolog	

Figure 3.5 – Protein homology page for the CitB protein. At the top is a brief description of the analysis, containing details on the FASTA alignment tool and on the table. The table is composed of 5 columns: Organism – lists all 16 organisms used; Protein name – best hit protein found in the respective organism with a hyperlink to UniProt; Identity and Similarity – scores from FASTA alignment tool; Bidirectional best hit (Yes/ No) – result of the bidirectional alignment.

Finally, to complement the new protein homology section, the Cluster of Orthologous Genes (COG) database is now fully integrated in *Subti*Wiki. COG is a tool for comparative and functional genomics of prokaryotes and identifies orthologs in a representative set of different bacteria and archaea(Galperin et al., 2021). By taking advantage of the optimized COG database and fully integrating their IDs, *Subti*Wiki now provides a strong complementary section of homologs and orthologs (Figure 3.6).

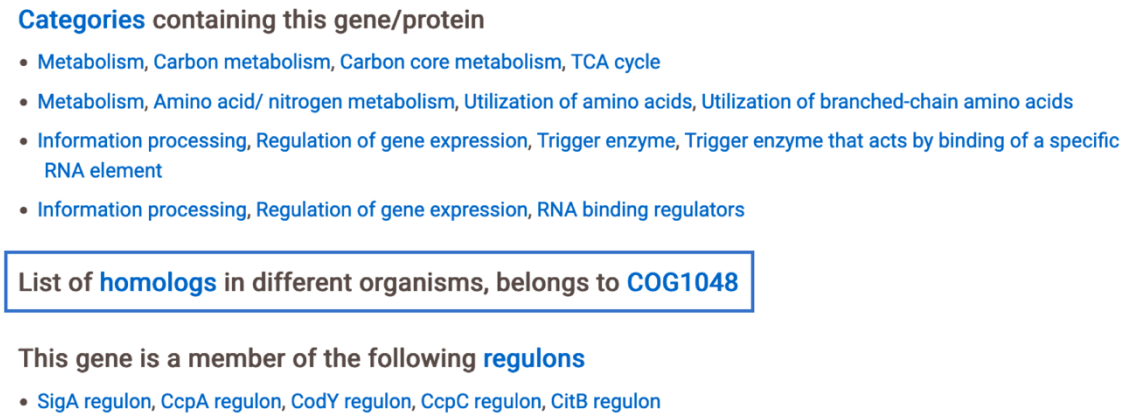


Figure 3.6 – Homology and COG database implementation in *Subti*Wiki on the *citB* gene page. The section highlighted in blue is shown when a corresponding COG entry is available for the gene and redirects to the respective COG database page.

3.6. New data integration

A hallmark of this *Subti*Wiki update is the integration of new data sets. Recently we have added a compendium of genome minimized strains derived from *B. subtilis* 168. Some of these strains have already been proven superior for biotechnological applications such as the production and secretion of difficult proteins and of lantibiotics (Suárez et al., 2019; van Tilburg et al., 2020). The MiniBacillus Compendium section (http://www.subtiwiki.uni-goettingen.de/v4/minibacillus) (Michalik et al., 2021) contains a table listing the name of each strain, the respective download link of Geneious and GenBank files, genomic details (genome size, deletion steps and percentage of genome reduction), as well as a list of publications associated with the respective strain.

Furthermore, we have added a large set of protein expression information. This information is based on a large-scale study that sought to compare proteomic responses to 91 different antibiotics and comparator compounds in an attempt to elucidate antibacterial modes of action (Senges et al., 2021). To supplement this representation, we also provide the possibility to download the entire dataset under the new Downloads section (http://www.subtiwiki.uni-goettingen.de/v4/download). In addition to the previously implemented expression data, the new data set can be accessed in the Expression Browser. Here, the user has full access to the available data for each biological element in a specific representation and it is possible to overlay the expression data for each protein in the Interaction Browser.

The latest updates reflect intensive research to identify novel transporters in *B. subtilis*. These include transporters for bicarbonate (NdhF-YbcC)(S. H. Fan et al., 2019; S.-H. Fan et al., 2021) and

several amino acids including alanine (AlaP), glutamate and serine (both AimA) as well as the toxic analogon glyphosate (GltT) (Klewing et al., 2020; Krüger et al., 2021; Sidiq et al., 2021; Wicke et al., 2019).

In addition, two new categories of genes/proteins have been included. These are the quasi-essential genes and genes that encode proteins that are required for the detoxification of toxic metabolites. Quasi-essential genes are those genes whose inactivation results in the immediate acquisition of suppressor mutations. Interestingly, essentiality of such quasi-essential genes has so far been controversial (Commichau et al., 2013; Figaro et al., 2013; Koo et al., 2017). The new concept of quasi-essentiality supports the idea that often there is no clear binary answer in this issue but rather a continuum from clearly dispensable genes, genes that prove essential only after long-term cultivation, genes that require genetic suppression upon deletion to those that are cannot be deleted under any circumstances. As an example, the *topA* gene encoding DNA topoisomerase I is responsible for the relaxation of negatively supercoiled DNA behind the RNA polymerase. Upon deletion of *topA*, the bacteria immediately acquire mutations resulting in an increased expression DNA topoisomerase IV which can then take over the role of TopA (Reuß et al., 2019). Similarly, deletion of the *rny* gene encoding RNase Y results in the acquisition of suppressor mutations that result in a globally reduced level of transcription activity (Benda et al., 2021) and deletion of the unknown gene *yqhY* triggers mutations that often affect conserved residues of acetyl-CoA synthetase indicating that YqhY prevents a potentially toxic accumulation of malonyl-CoA or of fatty acids (Tödter et al., 2017). Proteins involved in detoxification reactions have recently been recognized as very important for cellular metabolism as they remove harmful by-products or intermediates. Examples for such toxic by-products are 4-phosphoerythronate, a by-product of erythrose-4-phosphate oxidation in the pentose phosphate pathway that inhibits the phosphogluconate dehydrogenase GndA, and 5-oxoproline, an unavoidable damage product formed spontaneously from glutamine. These toxic metabolites are disposed of by the CpgA GTPase which moonlights in dephosphorylation of 4-phosphoerythronate(Niehaus et al., 2017; Sachla & Helmann, 2019) and by the PxpABC 5-oxoprolinase (Niehaus et al., 2017), respectively. It has recently become obvious that these detoxification mechanisms are very important for the viability of any living cell. The limited knowledge on these mechanisms is an important bottleneck in all genome reduction projects (Reuß et al., 2016, 2017). Thus, both these new categories are very important to get a more comprehensive knowledge about the minimal requirements to sustain bacterial life.

3.7. Future perspectives

With millions of accesses on a yearly basis, *Subti*Wiki has become one of the most popular database that provides up-to-date and curated data for the model organism *B. subtilis*. Although this bacterium is one of the best-studied organisms, there is still a vast portion of data to be annotated and a lot to be discovered. By providing daily data and feature updates to the community, we expect *Subti*Wiki to continue to grow. We hope for *Subti*Wiki to expand in viewership and further establish itself as a prominent tool helping researchers develop new hypotheses. Future aims include the addition of new basis entities such as metabolites and corresponding new types of data sets such as protein-metabolite, protein-RNA, and RNA-metabolite interactions that will consolidate the role of *Subti*Wiki as a trendsetter for model organism databases.

Acknowledgements

We are grateful to all lab members for testing the new features and for valuable feedback.

Conflict of Interest

The authors declare that they have no conflicts of interest with the contents of this article.

Chapter 4 – A relational database for the synthetic organism JCVI-syn3A

The results described in this chapter were originally published in *Protein Science* (doi: 10.1002/pro.4179):

SynWiki: Functional annotation of the first artificial organism Mycoplasma mycoides JCVI-syn3A.

Tiago Pedreira[1], Christoph Elfmann[1], Neil Singh[1] and Jörg Stülke[1]

[1] Department of General Microbiology, GZMB, Georg-August-University Göttingen, Germany

Author contributions

Tiago Pedreira: Conceptualization (lead); investigation (lead); methodology (lead); project administration (supporting); resources (lead); supervision (lead); validation (equal); visualization (equal); writing; original draft (equal); writing; review and editing (equal). Christoph Elfmann: Methodology (supporting); resources (supporting); visualization (supporting). Neil Singh: Conceptualization (supporting); data curation (supporting); resources (supporting). Jörg Stülke: Conceptualization (equal); data curation (equal); funding acquisition (lead); project administration (lead); supervision (lead); writing; original draft (equal); writing; review and editing (equal).

4.1. Abstract

The new field of synthetic biology aims at the creation of artificially designed organisms. A major breakthrough in the field was the generation of the artificial synthetic organism *Mycoplasma mycoides* JCVI-syn3A. This bacterium possesses only 452 protein-coding genes, the smallest number for any organism that is viable independent of a host cell. However, about one third of the proteins have no known function indicating major gaps in our understanding of simple living cells. To facilitate the investigation of the components of this minimal bacterium, we have generated the database *Syn*Wiki (http://synwiki.uni-goettingen.de/). *Syn*Wiki is based on a relational database and gives access to published information about the genes and proteins of *M. mycoides* JCVI-syn3A. To gain a better understanding of the functions of the genes and proteins of the artificial bacteria, protein-protein interactions that may provide clues for the protein functions are included in an interactive manner. *Syn*Wiki is an important tool for the synthetic biology community that will support the comprehensive understanding of a minimal cell as well as the functional annotation of so far uncharacterized proteins.

4.2. Introduction

The creation of artificial cells that contain only those genes that are essentially required to sustain the major cellular functions is one of the aims of the emerging field of synthetic biology. This aim can be achieved in two complementary approaches: the top-down approach identifies essential genes and functions, predicts the minimal gene set, and deletes all non-essential genes in a consecutive manner thus resulting in genome reduced strains. This approach has so far been most successful for the Gram-positive model bacterium *Bacillus subtilis*. For this bacterium, the essential gene set has been identified and the genes required for a minimal genome have been predicted (Commichau et al., 2013; Koo et al., 2017; Reuß et al., 2016) and the genome has been reduced by about 40% (Reuß et al., 2017). Alternatively, in the bottom-up approach genes predicted or identified as components of a minimal genome are synthesized *in vitro*, assembled and introduced into a living cell. The original DNA will then be lost, and the cellular activities will be driven by proteins encoded by the artificial DNA molecule. This approach has been used to synthesize *Mycoplasma mycoides* JCVI-syn3A, an artificial minimal bacterium with as few as 452 protein-coding genes, less than in any other known natural independently viable bacterium (Breuer et al., 2019; Hutchison et al., 2016). Of the proteins encoded by *M. mycoides* JCVI-syn3A, about one third has no known function, indicating how far we still are from a comprehensive understanding of even a very simple and minimal living cell. The investigation of *Mycoplasma mycoides* JCVI-syn3.0 by the scientific community will certainly help to get a deeper

appreciation for the principal requirements of a minimalistic form of life, and thus touches one of the most basic and most important aspects of biology.

To facilitate the investigation of *Mycoplasma mycoides* JCVI-syn3A, we have developed *Syn*Wiki, a database centered around the genes and proteins of this bacterium. This database is based on the platform of the database *Subti*Wiki which is designed for the functional annotation of *B. subtilis* (Michna et al., 2014; Zhu & Stülke, 2018). *Syn*Wiki presents the available information on the genes and proteins of *Mycoplasma mycoides* JCVI-syn3A in an easy and highly intuitive manner. One focus of *Syn*Wiki is the presentation of links between different genes and proteins that allow the researcher to develop novel hypotheses. The information provided is based on the publications describing the construction of the organism and the reconstruction of the minimal metabolism (Breuer et al., 2019; Hutchison et al., 2016) and on the published specific information available on the proteins.

4.3. Description of the database

*Syn*Wiki is a relational database for the functional annotation of the synthetic minimal microorganism *Mycoplasma mycoides syn3A* and uses the data describing the original JCVI-syn3.0 bacterium as well as the most recent iteration of this organism, JCVI-syn3A (Breuer et al., 2019; Hutchison et al., 2016). When referring to *Syn*Wiki data, we always consider the latter organism.

In *Syn*Wiki, the central biological element is the gene with its underlying relationships to proteins and functional RNAs of JCVI-syn3A. With this in mind, it is clear that most pages focus on a specific gene element and its functional annotation. To access any of these elements, the front page provides a search box and access to several features of different types of data (Figure 4.1). To view information on any gene, it is only necessary to query for a specific gene's identifier (see more below). Alternatively, it is possible to go directly to a random gene page using a button on the top of the page. In addition to providing scientific information, *Syn*Wiki also serves as a hub for the scientific community interested in the minimal organism. On each gene page, there is a banner that gives important information such as upcoming conferences and other events, and a link to labs working on this organism.

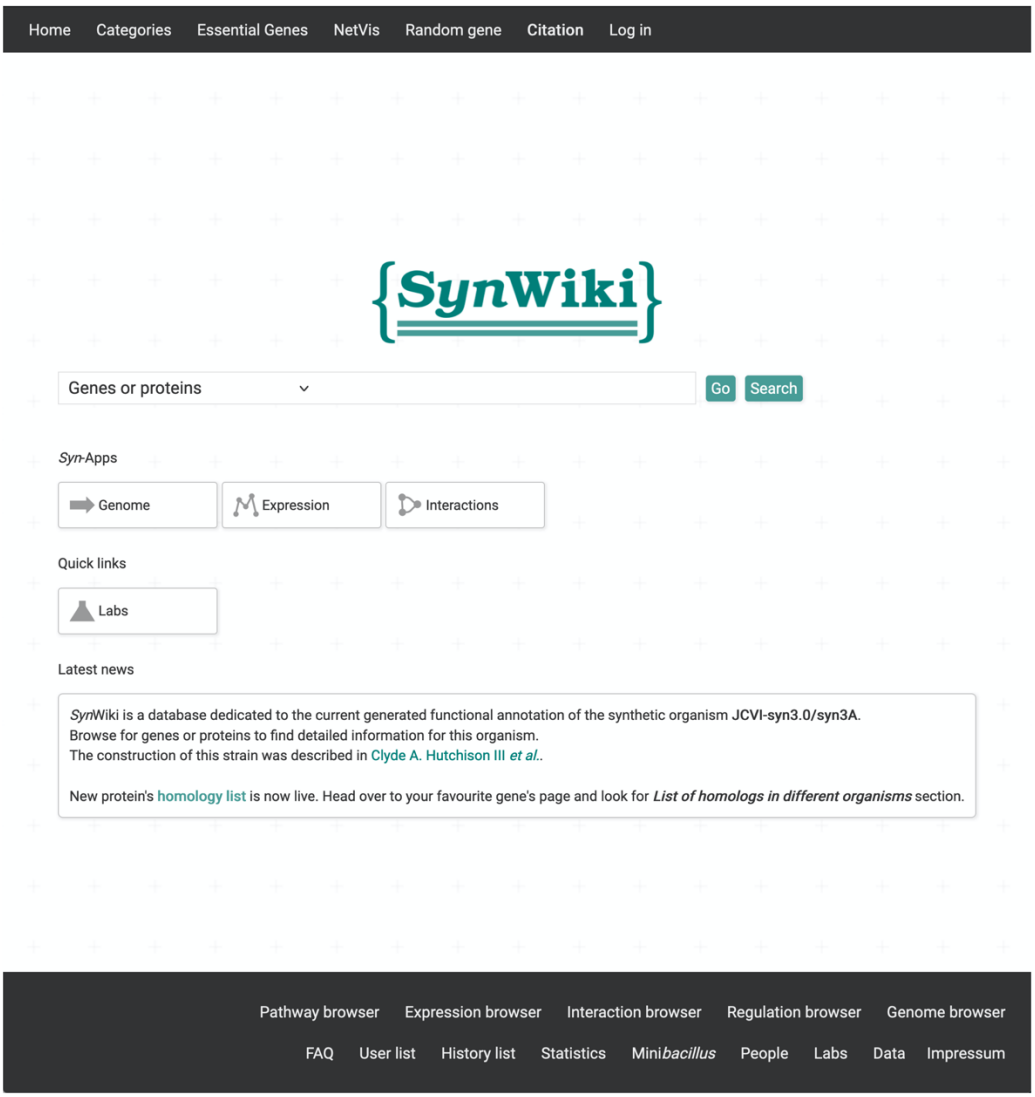


Figure 4.1 – Home page of *Syn*Wiki. The main element of this page is the search bar that allows for searching of biological elements. Further down there is a list of *Syn*Apps and other helpful links such as a list of labs that work with this organism.

4.4. *Syn*Wiki identifiers

Currently, two unique identifiers can be used for each gene. The first one is a specific gene/ protein name that serves as a mnemonic for its function (such as *eno* for enolase) or its synonym (such as *ktrC* with its synonym *trkA*). The second identifier is an "eternal" locus tag, a gene-specific identifier that will remain unchanged even though mnemonic gene designation may alter (Breuer et al., 2019). For this, we have maintained the original locus tags created for JCVI-syn3A, which abide by the following rules: a "JCVISYN3A_" prefix with a numerical suffix "XXXX" that derives from the initial data (Hutchison et al., 2016). As an example, to retrieve information on the JCVISYN3A_0001 gene, or *dnaA*, it is

possible to search for either the locus tag or gene name. For uncharacterized genes, that yet lack a mnemonic designation, such as JCVISYN3A_0399, only this locus identifier can be used to access the page.

4.5. The gene pages

As mentioned before, *Syn*Wiki revolves around the gene entity, meaning that each gene has a fully dedicated page where all documented information can be found. Independent of the data available for each gene, all gene pages share the same structure (Figure 4.2). At the top, there is a banner highlighting community events, just followed by the gene name accompanied and a short description of its function, if available (Figure 4.2A). Followed by this, there is a table containing specific information of the searched gene and its product (Figure 4.2B). It contains information regarding locus tag, protein specific information (molecular weight, isoelectric point, product and function), gene and protein lengths with direct links to their corresponding sequences., a BLAST search, and information on essentiality. Moreover, *Syn*Wiki provides links to the respective entries in UniProt, KEGG, KEGG Pathway, and STRING databases (Figure 4.2B). Below that, an interactive genomic context browser is shown, which is part of the corresponding Genome Browser *Syn*App (see more below) (Figure 4.2C). This feature allows to see the gene's position in the genome and to scroll through the genome. Right under this section, information on the annotated functional categories (see below), as well as a link to the protein homology analysis developed in our lab (see below) are provided (Figure 4.2D). The remaining part of the page presents additional information on the gene/ protein. The gene section lists information regarding the gene element, such as genomic coordinates, phenotypes of mutants if available and other information centered on the gene (Figure 4.2E). The protein section lists specific information regarding this element, such as protein family and biochemical details (Figure 4.2F). At the bottom there is a list of publications annotated for the current biological element (Figure 4.2G). Although not currently represented in *Syn*Wiki due to lack of data, information on operons and regulations can also be displayed in this page. Finally, the right panel of the page provides links to other important pages such as a list of labs working on JCVI-syn3A, search boxes for *Syn*Wiki and PubMed entries, and displays protein structures as well as protein-protein interactions (if any) (Figure 4.2H).

Figure 4.2 – Gene page overview for *ptsH*. Gene page structure is shared among all genes with differences in displayed data, as it is dependent on its availability. A – Gene title and description; B – Overview table focusing on different gene/protein details; C – Genomic context of genes with an interactive genome browser; D – List of functional categories the gene is annotated with and list of homologs in other organisms; E – Section focused on gene centered information; F – Section focused on protein centered

information; G – List of references annotated to the gene; H – Overview of protein structure and protein-protein interactions.

4.6. *Syn*Apps

One of the strong points of our family of databases is the integration of different levels of data using multiple browsers under "*Syn*Apps" (Figure 4.1). Among these, we highlight the integration of genomic context (Genome Browser), expression data (Expression Browser), and protein-protein interactions (Interaction Browser).

In the Genome Browser, the user gets access to a scrollable genome to check for gene context and orientation, as well as DNA and protein sequences. As mentioned before, this browser is also partially included in every gene page for easier overview of each gene's genomic context. Importantly, due to evolutionary pressure towards an AT rich genome, Mycoplasmas use the TGA codon for tryptophan, in contrast to other organisms that use it as a stop codon instead (Inamine et al., 1990; Yamao et al., 1985). Therefore, the internal codon table was adjusted accordingly, and all protein sequences displayed in Genome Browser are correctly translated.

The Expression Browser provides data on a proteomics analysis during early growth for 428 proteins (Breuer et al., 2019). These data are also presented on the gene pages (in the paragraph "Expression and Regulation"). However, the Expression Browser allows a direct comparison of the protein amounts for different proteins of choice (see Figure 4.3A).

To identify the functions of so far unknown or poorly studied proteins, data on physical protein-protein interactions are of key importance. We have recently used a combination of cross-linking, mass spectrometry and cryo-electron tomography to unravel the *in vivo* interactome of *Mycoplasma pneumoniae* (O'Reilly et al., 2020). While global interaction data are not yet available, a recent study has shed light on the prediction of complex interactions in *M. mycoides* JCVI-syn3A(Zhang et al., 2021). Protein-protein interactions are displayed on the gene pages (Figure 4.2H) and in the Interaction browser (Figure 4.3B). Similar to the Expression Browser, the Interaction Browser provides opportunities of interrogation that could not be reached on the gene pages. As an example, the user can visualize complex interaction patterns and selection one protein highlights direct neighbours and displays an overview from the gene page (Figure 4.3B).

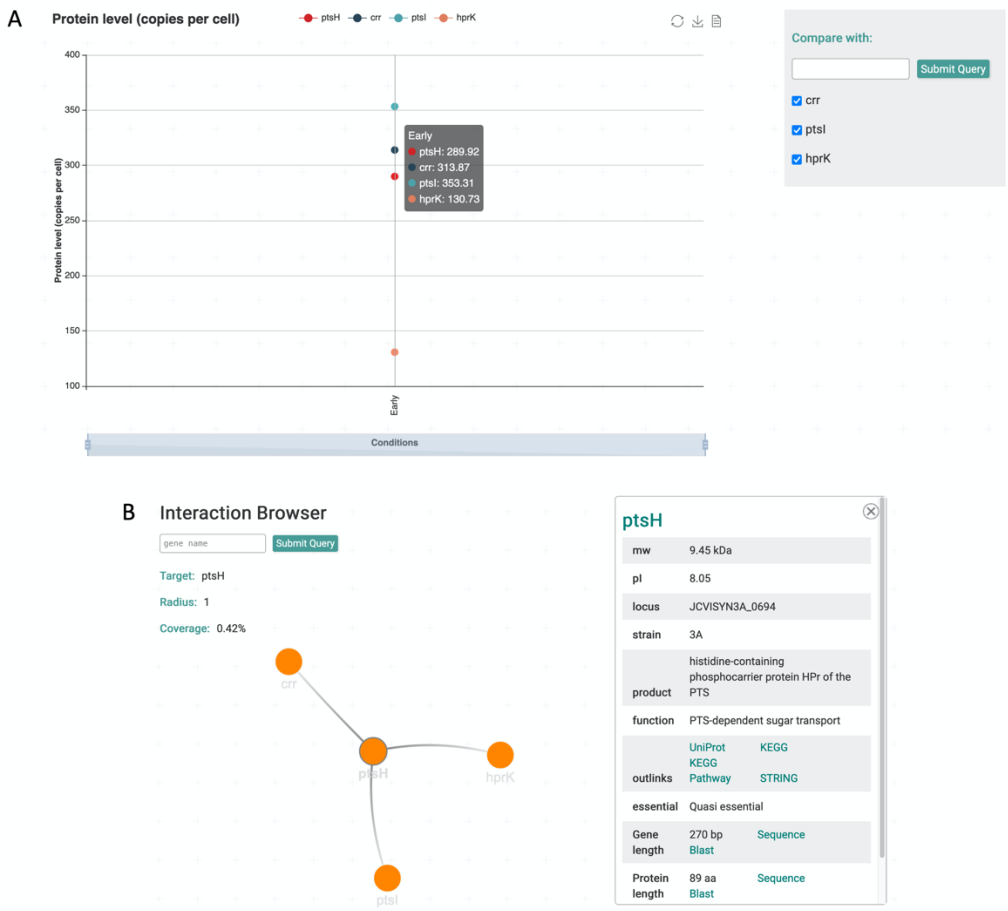


Figure 4.3 – Overview of implemented *Syn*Apps. A – Expression Browser on protein level for PtsH. Shows the number of protein molecules per cell under each experimental condition. Only "Early growth phase" data is available, thus it is the only one being represented. It is possible to add other proteins for comparison on the same view (all interacting proteins shown, Crr, PtsI, and HprK) and it is also possible to download data directly. B – Interaction browser for PtsH, highlighting direct interaction partners and displaying an overview from its gene page.

4.7. Implementation of the database

4.7.1. *Syn*Wiki architecture

*Syn*Wiki's architecture relies on the framework previously developed for the well-known database *Subti*Wiki (Zhu & Stülke, 2018). The framework uses a relational database management system to establish relationships between entities, based on the relation theory(Codd, 1970), thus allowing the organization and storage of data in tables. This allows for the creation of complex and rich relationships between genes/proteins and their annotation. Another advantage of using *Subti*Wiki's framework is

that it provides the full experience one can find in this database, such as Browser features and the fully-fledged user system to add and edit data.

4.7.2. *Syn*Wiki data

As mentioned before, *Syn*Wiki uses the data from the original JCVI-syn3.0 and JCVI-syn3A publications (Breuer et al., 2019; Hutchison et al., 2016). According to the authors, the latter iteration of this organism features the addition of 17 genes (14 *Mycoplasma mycoides* genes and 3 pseudogenes), making a total of 492 genes, resulting in better growth and improved stability.

In addition to the annotation in the original publications we also went through the literature, checked known functions of homologs, and searched for the availability of protein structures of the JCVI-syn3A proteins or their homologs from other organisms. All these results from manual data curation were added to the pages. Moreover, the curation of the pages is an ongoing process that will lead to ever-improved annotation and data presentation.

The success of a biological database strongly depends on the quality of annotation, and for those cases where there is poor or no annotation, we wanted to take a step forward and give our own contribution. With this in mind, and starting from the original annotation, we have also assigned each gene to one or more functional category. We took inspiration from GO term tree (Ashburner et al., 2000) and created a tree-like structure containing all functional categories. Currently, there are four major functional branches, “Cellular processes”, “Group of genes”, “Information processing” and “Metabolism”, which then branch out into more specific categories (see here for an overview: http://synwiki.uni-goettingen.de/v1/category, and Table 1). For example, in “Cellular processes” it first branches out into “Cellular envelope and cell division”, “Homeostasis”, and “Transporters”. Then, “Homeostasis” branches out into “Metal ion homeostasis” and “Coping with stress”; “Transporters” branch out into “ABC transporters”, “ECF transporters”, “Phosphotransferase system” and “Symporter”. The “Group of genes” parent category branches out into “Foreign genes”, “Membrane proteins”, “Poorly characterized enzymes”, “Proteins of unknown function”, “Pseudogenes”, “ncRNAs” and “Essentiality”. As a result of this annotation, we can now dig deeper into the data and assess it according to specific categories. As an example, looking for “Essentiality” allows to identify 46% of genes to be labeled as “Essential”, while only 13,6% are classified as “Non-essential” (Figure 4.4). Moreover, we have annotated most genes with over 500 publications to help better understand the potential underlying roles of JCVI-syn3A genes.

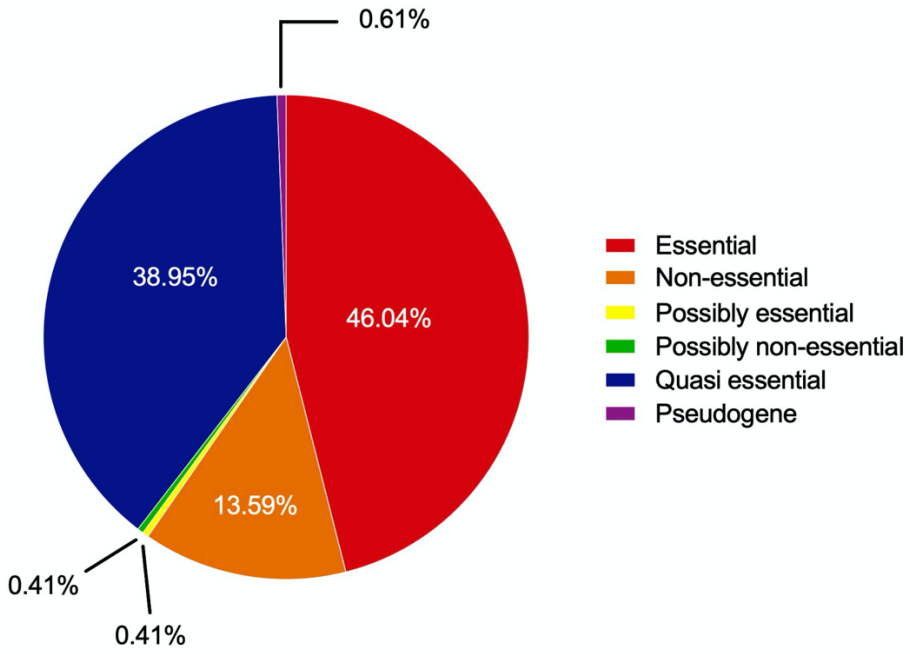


Figure 4.4 – Distribution of essentiality among all genes in *Syn*Wiki. While essential genes cannot be removed from the genome without loss of viability, the removal of quasi-essential genes results in an observable growth disadvantage (Breuer et al., 2019).

Additionally, we added an extensive list of potential protein homologs for 16 relevant bacterial species, among them *M. pneumoniae* and the model organisms *Escherichia coli* and *B. subtilis*. The proteomes of these organisms were extracted from UniProt (Bateman et al., 2021) and a set of a bidirectional alignments was performed using the similarity search tool FASTA (Madeira et al., 2019) between the library of proteomes and the proteome of JCVI-syn3A. The resulting aligned protein pairs in both directions showing an E-value ≤0.01 and high similarity were considered to be potential bidirectional homologs. The results can be accessed on the gene page (Figure 4.2D) and they are presented in detail in a table format, showing the respective scores (Figure 4.5). To complement this, and based on protein structure homology, we have included a protein structure prediction for 327 proteins (Figure 4.2H).

PtsH

Organism	Protein name	Identity	Similarity	Bidirectional best hit
Mycoplasma mycoides subsp mycoides	PtsH	96.6%	100.0%	Yes
Mycoplasma genitalium	PtsH	54.4%	87.3%	Yes
Mycoplasma pneumoniae	PtsH	46.3%	86.6%	Yes
Mesoplasma florum	Mfl565	57.3%	88.8%	Yes
Acholeplasma laidlawii	Hpr1	45.1%	76.8%	Yes
Bacillus subtilis	PtsH	44.4%	81.5%	Yes
Listeria monocytogenes	PtsH	42.0%	80.2%	Yes
Streptococcus pneumoniae	PtsH	46.9%	81.5%	Yes
Clostridium acetobutylicum	CA_C1820	33.8%	76.6%	Yes
Corynebacterium glutamicum	BAB99330	30.9%	75.3%	Yes
Streptomyces coelicolor	PtsH	26.4%	65.3%	Yes
Escherichia coli	PtsH	38.2%	78.9%	Yes
Prochlorococcus marinus		No significant homologs	No significant homologs	
Synechococcus elongatus		No significant homologs	No significant homologs	
Synechocystis sp		No significant homologs	No significant homologs	
Borrelia burgdorferi	PtsH	38.0%	75.9%	Yes

Figure 4.5 – Representation of protein homology analysis for PtsH. Results displayed in a table format for each organism used in this analysis with respective best potential homolog with UniProt link. Identity, similarity and bidirectionality scores also displayed.

4.8. Future perspectives

*Syn*Wiki is a new database for the recently created minimal genome microorganism *M. mycoides* JCVI-syn3A. *Syn*Wiki uses the framework of *Subti*Wiki, which gives it a robust and consistent way of searching and updating data. Although there is scarce information for this organism, we have created a powerful structure to store and display the current data. However, *Syn*Wiki is also prepared to include new findings that might arise from emerging studies. Importantly, recent attempts to model the complete metabolism of JCVI-syn3A (Breuer et al., 2019) and related natural minimal organisms *Mesoplasma florum* (Lachance et al., 2021) will certainly benefit from the collection of information

provided in *Syn*Wiki. On the other hand, the information derived from the metabolic models and similar global approaches is important to update *Syn*Wiki. As more research groups delve into the unknown and try to unveil more of JCVI-syn3A, we want to provide the scientific community with a platform where all biological elements of this organism can be updated on a daily basis with curated data. We aim not only to put data together, but also to organize it and give scientists the confidence and the necessary tools to create new approaches for their research.

Acknowledgements

We are grateful to Hannes Gebauer for the initial work on *Syn*Wiki.

Conflict of interest

The authors declare that they have no conflicts of interest with the contents of this article.

Chapter 5 – *Core*Wiki, a novel framework for prokaryotes

5.1. Abstract

In a way to address the increasing amount of information generated yearly by novel techniques and with the emergence of new types of data, biological databases play a fundamental role in organising this data. Among these data platforms, *Subti*Wiki stands out as the golden standard for all researchers working on *B. subtilis*. However, this database is also subject of improvement as it lacks the proper documentation to aid the development of further features. *Core*Wiki is a new framework that aims to replace the current live iteration of *Subti*Wiki v4. *Core*Wiki was developed with Flask, a micro-framework written in Python that is remarkable for its flexibility, allowing to embody the traditional design pattern of Model-View-Controller with an easy-to-implement core feature. Additionally, an improved database schema that uses *Subti*Wiki v4 database as a comparison term was implemented. New models were added to accommodate the bacterial data, retaining the current relationships, and created novel ones to further expand the information complexity. Moreover, the new schema is further explored by adding new interaction models and retrieving long forgotten sections from an early *Subti*Wiki iteration, showing how flexible the presented database and framework can be. *Core*Wiki inherits the current design of *Subti*Wiki, perfectly integrating the new data models. As a final goal, *Core*Wiki aims to be the chosen framework by many research groups when it comes down to integrating bacterial information.

5.2. Introduction

In this digital era, where high throughput data techniques reign supreme, it is of utmost importance to have specialized structures to handle this data. To support this endeavour, scientists count with numerous biological databases that aim to store and represent high amounts of information (Alkan et al., 2011; Caswell et al., 2019; Loman et al., 2012; Manzoni et al., 2018; Mardis, 2017; Reuter et al., 2015). These databases are particularly useful not only for their storage capabilities, but also due to their feature to integrate multileveled information. However, setting this data on the same plane as all the inherent functional annotation of a certain biological element represents a major hurdle. Information from NGS, proteomics, transcriptomics, metabolomics and interaction between elements are classified as the main players in the increase of data complexity (Corley et al., 2020; Gerovac et al., 2021; Link et al., 2013; Loman et al., 2012; Mardis, 2017; O'Reilly et al., 2020).

Biological databases play the fundamental role to take all this complex and high variety data and give it an important biological context (Baxevanis & Bateman, 2015; Caswell et al., 2019). While some of these structures may provide exceptional support on specific levels of data nature, e.g., protein or gene information only, and regardless of the organism, there are other databases that are dedicated to a single model organism, i.e., model organism databases. Among these, *Subti*Wiki is the most accessed biological database for the model organism *B. subtilis* (Zhu & Stülke, 2018; Pedreira et al., 2022). This database provides curated information for this organism while providing with an effective way of keeping up with the current data generation.

Under the hood, the most recent iteration of *Subti*Wiki, version 4 (Pedreira et al., 2022), is organised in two different sections: server-side and client-side, also known as frontend and backend, respectively (Figure 5.1). *Subti*Wiki counts with a frontend composed by Hypertext Markup Language (HTML) (Hickson et al., 2022; Karan, 2013), responsible for the generation of the structural elements of the website, Cascading Style Sheets (CSS) (Lie & Bos, 2005), used for styling of each element of the webpage, JavaScript (JS) (Flanagan, 2011) to manipulate web content and how it behaves, and AJAX to enable asynchronous data exchange. For the backend, this platform relies on a web server Apache, to enable data exchange between frontend and backend, PHP (Bakken et al., 2000) that serves as a server-side scripting language, and a database management system (DBMS) that is used to access, create, update or delete data, which in this case is MySQL (Axmark & Widenius, 2022).

Both ends orchestrate together the communication between users and the database in what is called the Linux Apache MySQL PHP (LAMP) stack (Figure 5.1). This structure enables the communication between both ends in a phased manner, for example if the user requests on the frontend to access data on a certain gene, the following happens:

- User requests information on a gene (Figure 5.1 A)
- The request is sent via HTTP by the browser and received by the web server (Figure 5.1 B)
- The request is sent to the database and PHP decodes the language received from the web server and translates it to SQL (Figure 5.1 C)
- The database sends a response to the web server and PHP decodes the response back for the end user's understanding (Figure 5.1 D)
- The web server sends the decoded response to the browser (Figure 5.1 E)

Upon receiving the information HTML, CSS and JS will act on their own and make sure the structure, styling and behaviour are applied properly.

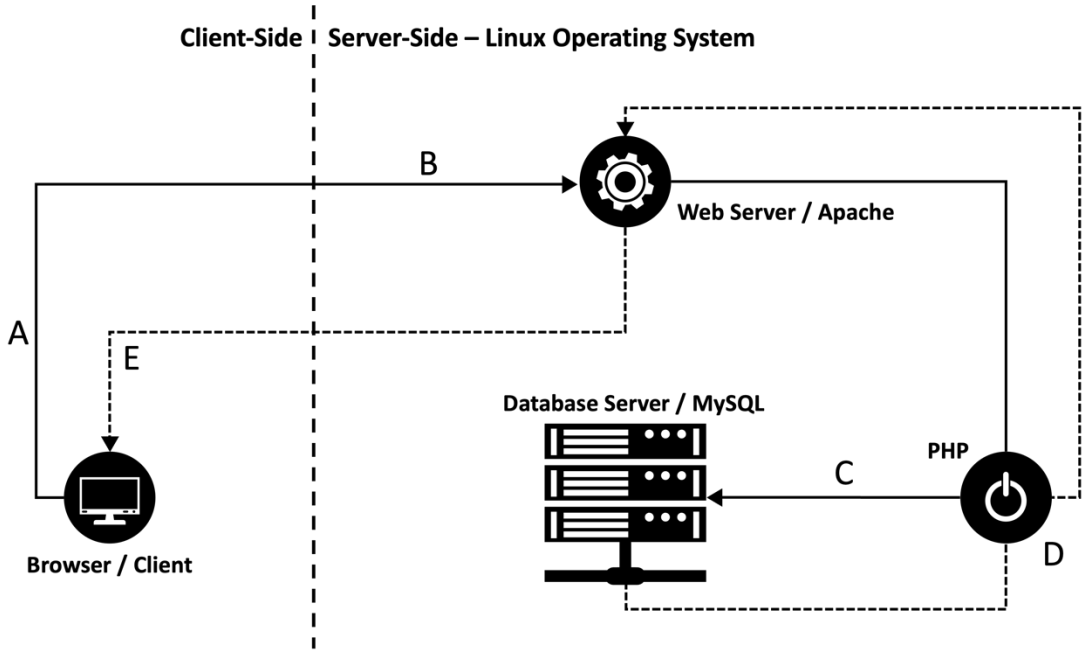


Figure 5.1 – Scheme of *Subti*Wiki v4 architecture. LAMP stack includes two types of actions: request (solid lines, ——) and response (dashed lines, ----). An example of how the stack handles information in SubtiWiki is as follows: A – Request from the client to the server-side. B – HTTP request handled by Apache and forward to the database. C – PHP decodes the request sent from Apache to SQL. D – Database responds with result from query in SQL, which PHP decodes and redirects to the web server. E – Apache retrieves the response and sends it to the client, where the user is now presented with the result of the query.

One of the major features of this framework is the possibility to expand beyond *B. subtilis*, adapting this structure to different organisms. With solid evidence of reusing successfully the same framework for *M. pneumoniae* (*Myco*Wiki) and *L. monocytogenes* (*Listi*Wiki), this structure contains, however, fundamental flaws regarding its development and maintainability. *Subti*Wiki's framework was tailor made (Zhu & Stülke, 2018) to specifically accommodate *B. subtilis* information, providing custom features that were designed with this organism in mind. As most prokaryotes share some degree of similarity, scaling beyond one organism is feasible. For this, it is extremely necessary to have the proper blueprints of a framework to be able to successfully adapt from one organism to the other. While it is easier if the developer in charge is responsible for the scaling, once more developers are added to the question, then the proper documentation will play a fundamental role in guiding new developers.

A software's documentation enables the addition of novel features by providing the necessary guide to each building block of the framework. Additionally, by having full control over the framework, it is easier to perform maintenance on the structure. According to ISO/IEC 25010:2011 Systems and software Quality Requirements and Evaluation (SQuaRE), **Maintenance** is an importance category of a

software's **Product Quality**, which contain characteristics that are important for every developer (Estdale & Georgiadou, 2018). This is where *Subti*Wiki's framework is at fault, as there is a lack of documentation, which massively hinders the development of novel features, regardless of the organism.

A potential approach to address this challenge is to develop a new platform, using an existing and well-established open-source framework. This will help developers by providing native documentation and support throughout each component. Moreover, most frameworks contemplate with implemented routing tables, which makes a big part of the heavy lifting of routing an easy task. Importantly, this requires the evaluation of which suitable framework would work the best for the project and which structural improvements would benefit the database. Thus, all these ideas are taken into consideration and a novel framework is introduced, called *Core*Wiki, that will aim to provide a generic platform for prokaryotes. For this, it is expected for this novel framework to replace all other Wiki's backend structure, and a thorough revision and overhaul of the database is made using *Subti*Wiki as a comparison term to evaluate the introduced modifications.

5.3. Methods and tools

5.3.1. Website Structure

Websites are divided in two sections: the backend and frontend. As discussed before, each part will play a role in how the whole framework will handle and represent data for the user. Currently, *Subti*-Wiki's framework runs on a LAMP stack (Figure 5.1), and the alternative architectures for *Core*Wiki will be discussed. On a functional level, the traditional request-response architecture found in *Subti*Wiki v4's framework will persist and thus, *Core*Wiki is planned to work in a similar way. There are, however, some changes planned to occur on the framework and database sides that aim to address the fundamental development flaws that the current platform presents. More specifically, CoreWiki will count with Flask (Grinberg, 2018) framework for the backend logic, rely on SQLite (Hipp, 2020) as a DBMS, and use SQLAlchemy (Bayer, 2012) as an Object-Relational-Mapper (ORM).

5.3.2. Database

5.3.2.1. Architecture of Databases

Biological data's integrity depends heavily on the relationships of a certain biological element. For example, the connection between a certain gene and its functional annotation can be mapped as a relationship between these two elements. To address this, *Subti*Wiki uses the relational database model

(RM) (Codd, 1970), where data is managed using a structure that allows relationships to be established. Data can be accessed, edited, created, and deleted using queries, where the user clearly states which data from which database to manipulate. This is handled by the DBMS, which will systematically organise the data under the user's rules, which in turn, are defined using Structured Query Language (SQL) (Axmark & Widenius, 2022).

In a relational database model, the **schema** defines an abstract design of its structure, where both the organisation of the data and the relationships are represented. Data is organised in tables, which are a visual representation of a relation. Inside each table, columns describe attributes of an instance of the data, and rows are instances of the relation presented in the form of a tuple. Each column of the table will have a set of constraints associated to the data type it can allow. For example, forcing a gene name to always be a variable character, **VARCHAR**, which can include variable length of numbers, letters, and special characters, or forcing an identifier to always be an integer, **INT**, or simply declaring which attribute is set to be unique across all instances, e.g., gene "ID" needs to be set to **UNIQUE** to avoid duplications.

Relationships can be established between tables and in order to maintain the integrity of the data and their relationships, the RM defines **keys** which are assigned to specific columns of each row of a table:

- **Primary key** – a special column that serves as the unique identifier of a whole table and each table cannot have more than one **primary key**
- **Foreign key** – a column or group of columns that serve as a cross-reference between two tables. This is the **primary key** of a different table, thus establishing a relationship between the two tables

Setting up these special keys to each table of the database model allows to set up relationships between different elements of different tables, creating one-to-one, one-to-many or even many-to-many relationships (Codd, 1970). In turn, this will increase the overall degree of complexity of the database, but also ensuring that higher levels of data can be easily modelled and integrated, without losing any information in the process.

Figure 5.2 shows an example of how keys can be used to establish relationships and maintain data's integrity. In this small **schema**, there are three tables: **Gene**, **Protein** and **Interaction**. By setting up a primary key to **Gene id**, **Protein** table can reference to it as a **foreign key**, establishing a relationship with the respective **Gene** entry. Thus, **Protein** table's **primary key** is the **foreign key** of **Gene** table. The same ideal can be applied to the **Interaction** table, which maps protein-protein interactions. By using as a **foreign key** on both **Protein_1** and **Protein_2** columns referring to **Gene id**, it is possible to

create a link between the data on all tables. In this case, by querying for **Gene** *gA*, the database accesses information regarding the genomic annotation (**Gene** table), protein properties (**Protein** table) and with which other proteins it can establish an interaction (**Interaction** table).

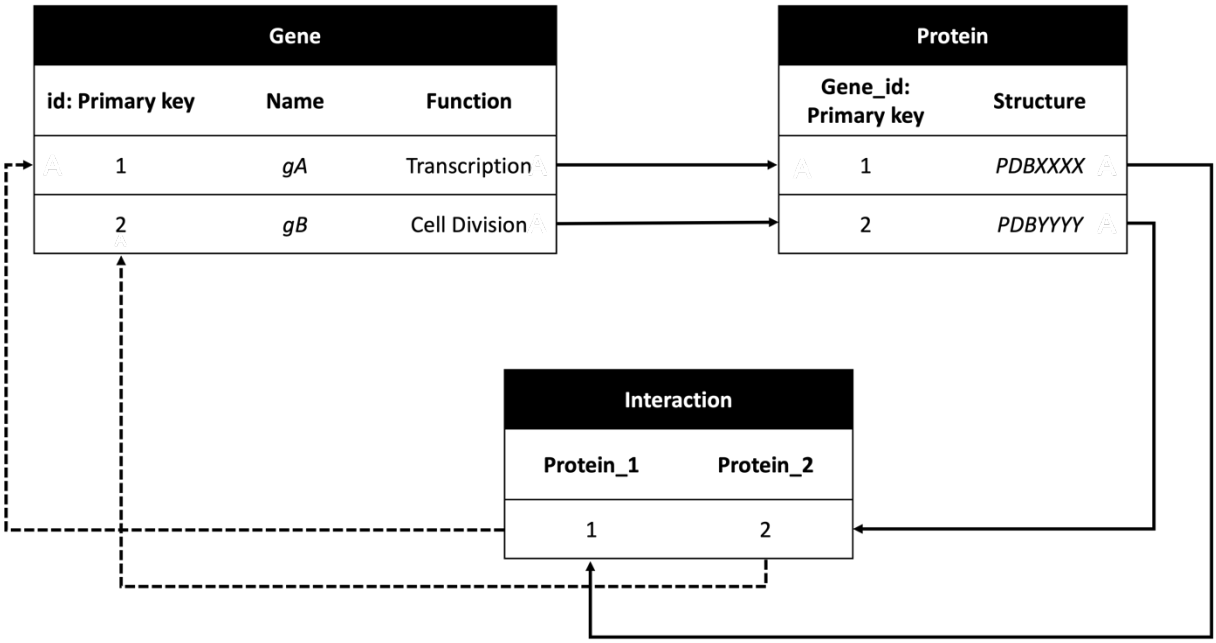


Figure 5.2 – Hypothetical example of biological data database **schema** containing a **Gene**, **Protein**, and protein-protein **Interaction** models. Solid lines (──) represent direct relationships, **Protein** table contains the **foreign key** to **Gene** table, allowing to establish the seen relationship. The same occurs between **Protein** and **Interaction** tables, which the latter contains the **foreign key** to the **Protein** table. In turn, the dashed lines (----) represent the indirect relationship **Interaction** and **Gene** table establish.

Since most programming languages operating on the backend of a website use **object-oriented programming** (OOP), where objects are the instances of **attributes** (referring to data) and **methods**, there is the necessity to map data generated in OOP when trying to convert it to a relational database model. Since SQL DBMS is not object-oriented, the data structure provided by OOP is by default incompatible with the one in the RM. To address this major constraint, backend frameworks are usually deployed with an Object-Relational Mapping (ORM), which serve as an object-oriented intermediary to convert OOP objects into a data structure readable by the relational database (Lorenz et al., 2017). This is achieved by creating an abstract layer between the user and the database, converting every OOP query or object manipulation into a language the database can read. The tables are never accessed directly, but rather by the ORM instead (Figure 5.3). Using this system enables to freely create data models in an OOP environment, which offers a lot of flexibility, while providing extra protection against undesirable injections of malicious SQL commands (Microsoft, 2021).

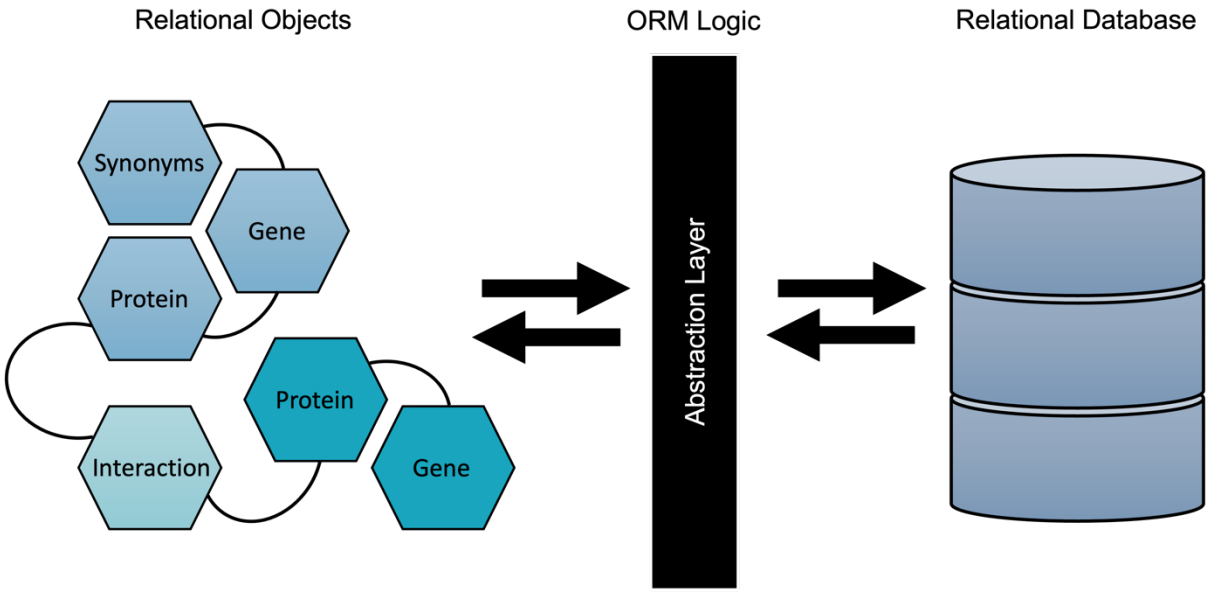


Figure 5.3 – Example of how instances of objects (hexagons, different colours represent different instances) and their relationships (solid lines, —) created by OOP interact with the abstraction layer provided by the ORM, which in turn allows these objects to be stored inside a relational database. Upon request, the ORM also handles the decode of database information back to the same instances of objects.

Accordingly, *Core*Wiki relies on the same principles of the ORM and RM **schema** to safely store information. This way, by enabling a strong and solid relationship model in the biological data, the construction of biological data models that can be opened up to any prokaryote is now possible.

5.3.2.2. SQLALchemy and ORM

As mentioned before, an ORM provides an object-oriented abstraction layer, allowing the creation of object-oriented data models, which can then be translated by the ORM into database language. *Subti*Wiki v4 makes full use of MySQL as DBMS, but since *Core*Wiki is built on Flask framework, SQLite (Hipp, 2020) as DBMS and SQLAlchemy as ORM have been chosen to handle the database as a whole. SQLAlchemy is an open-source SQL toolkit Python package, mostly known for its ORM that provides the data mapper pattern (Bayer, 2012). SQLAlchemy exposes all object relational details in a transparent way, allowing to automatize redundant tasks and to fully focus on the design of the database **schema**. Thus, the creation of model classes sets rules for the development of object attributes and relationships that will then populate the database in each table. As discussed before, each model will correspond to a different and independent table, that can establish relationships with other tables by

the means of **keys** (see more in section 5.3.2.1.). For this, Flask integrates SQLAlchemy in a very easy to understand way and for the sake of the example, Listing 5.1 shows the definition of two hypothetical model classes, **Gene** and **Protein**, as well as the relationship they establish between them.

```
1.   from flask_alchemy import SQLAlchemy
2.   db = SQLAlchemy()
3.
4.   class Gene(db.Model):
5.     __tablename__ = 'Gene'
6.     id = db.Column(db.String(40), primary_key=True)
7.     name = db.Column(db.String(255), nullable=False unique=True)
8.     protein = db.relationship('Protein', back_populates='gene')
9.
10.  class Protein(db.Model):
11.    __tablename__ ='Protein'
12.    id = db.Column(db.Integer, primary_key=True)
13.    gene_id = db.Column(db.String(40), db.ForeignKey('Gene.id'))
14.    structure = db.Column(db.String(255))
15.    gene = db.relationship('Gene', back_populates='protein')
```

Listing 5.1 – Example of two hypothetical model classes, **Gene** and **Protein**. Definition of each table's columns and constraints, such as how long each column may allow. The **primary** and **foreign keys** as well relationships are all set in the model.

This code snippet shows how to define model classes within the Flask-SQLAlchemy environment, where each class is considered to be a table, meaning that each will have a set of attributes and methods that define the properties and fields present in each table. Each variable inside the class will represent either a column, defined by **db.Column()**, or a relationship, declared by **db.relationship()**. As for the attributes, it is possible to define which type of data will be present in each column, for example in the **Gene** model, the ID is set to be composed of a 40 character long unique string. Relationships, on the other hand, do not represent a column, but a connection (line 8 in Listing 5.1). By declaring the variable **protein** to have a relationship, it is necessary to define which table it has a relationship with. In this case, setting the connection to the **Protein** table also requires defining which variable in the **Protein** model it establishes the relationship with, which is the variable **gene** (line 15 in Listing 5.1).

The second model, **Protein**, is declared and defined in a similar way as the previous one (line 10 in Listing 5.1). There, it is established that the column **gene_id** is the **foreign key** of column **id** from the **Gene** table (line 13 in Listing 5.1). This ensures that **gene_id** and **id** from **Gene** table are in a tight one-to-one relationship. This is not the case here, since a gene can only have one protein product, other types of relationship can be established, for example one-to-many (e.g., gene – synonyms) or many-to-many (e.g., genes – publications) relationships.

SQLAlchemy establishes all relationships automatically for the developer and makes it simple and easy to access information across tables, based on these relationships. To perform queries, the developer has access to a series of commands that are implemented functions from Flask-SQLAlchemy. For example, to query a single entry based on some criteria, let's say "id", from the previously defined **Gene** table, the developer can use the method **gene_of_interest = Gene.query.get(id)**. Querying every entry on the same table can also be done in a very easy and simplistic way, by using the command **Gene.query.all()**. Importantly, it is possible to instantly access all information from a certain entry and all the respective data it has a connection with from relationships established *a priori*. In the given example, to access all protein information from **gene_of_interest**, all it takes is to access the internal relationship of the object: **gene_of_interest.protein**.

To add new records to the database using this package is also made easy. Listing 5.2 shows the simulation of how to add manually a new entry in the **Gene** table, altogether with a relationship with the **Protein** table.

```
1.  from flask_alchemy import SQLAlchemy
2.  db = SQLAlchemy()
3.
4.  gene = Gene(
5.          id="90AFF93D6E7BC993B2773C12EE400C87H1T5B831",
6.          name = "geneA")
7.  gene.protein = Protein(structure='ABC')
8.
9.  db.session.add(gene)
10.  db.session.commit()
```

Listing 5.2 – Example of code to add a new entry in the **Gene** table as well as establishing a relationship with the **Protein** Table.

At the end of defining the data, running the commands **db.session.add()** and **db.session.commit()** (lines 9 and 10 in Listing 5.2) ensures that the data is properly saved in the database by SQLAlchemy. In the same sense, if there is the need to delete a record from the database, SQLAlchemy also contains a function to act accordingly, which is equally easy to use, **db.session.delete()**.

Finally, if the database schema requires to be updated, SQLAlchemy provides a tool to handle this. More specifically, SQLAlchemy Alembic allows to easily update the number of tables, columns of other aspects of the database without having to erase any records. Instead, by running the function **migrate()**, the database is updated in real time and all new settings are added/removed accordingly.

5.3.3. Backend Structure

5.3.3.1. Server side scripting language and working framework

As discussed before, in a website, the backend represents the server side, or data access layer (Figure 5.1). This means that in the backend of a framework it is possible to find the data storage handling as well as the whole logic to access and manipulate the data. Thus, all websites require a backend to work properly and *Core*Wiki will be no different.

In contrast with the current *Subti*Wiki's framework, *Core*Wiki uses Python as its server-side scripting language. Python is a popular, multipurpose, and high-level programming language that supports widely used programming paradigms, i.e., object-oriented, procedural, and functional programming (van Rossum & Fred L., 2009). In fact, Python scored as the top programming language used as of 2022, surpassing the previous leader language C and is used among many popular software companies (TIOBE, 2022). With a wider group of programmers, the community support for this language grows accordingly, making available a large number of open-source applications (van Rossum & Fred L., 2009). Due to the flexibility and wide support of this language among the scientific community, the choice to use Python as the server-side scripting language of *Core*Wiki was clear.

Having the language selected, the next step is to choose a compatible framework. Among some popular web development frameworks written in Python, Flask was chosen to be the suitable backend framework for *Core*Wiki (Grinberg, 2018). Flask is used among many well-known service platforms, such as Netflix, Reddit, Lyft and many more, and is considered to be a "micro web framework" (Grinberg, 2018; Stackshare, 2022). This classification of "microframework" is bound strictly to the fact that it merely uses minimal core functions, not making use of any particular tool or library. This, however, does not mean that Flask lacks in functionality, as it highly praises its extensibility (Grinberg, 2018). By not having any external library, the user is free to choose which other libraries or extension tools better suit their needs. Flask does not have a native database, database abstraction layer or even form validation, however, it offers support extensions that work as if they were native of Flask itself. Extensions such as an ORM, form and user validation will play an important role in the process of creating *Core*Wiki, as it is of utmost importance to have well-established user checkpoints when accessing and validating data from and to the database.

Flask is part of a collection of Python libraries designed for web development, called "The Pallets Projects" (Mönnich et al., 2016). The applications present in this collection were developed with the objective of using them altogether with Flask and since it is stripped off of a native ORM, user validation and even routing, these libraries aim to support the utility of this framework. Thus, for user-based platforms to work as intended, they rely on the following libraries:

- Werkzeug
- Jinja2
- MarkupSafe
- ItsDangerous

Werkzeug, meaning “tool” in the German language, is a comprehensive utility library for Web Server Gateway Interface (WSGI), or simply put, handling objects for requests, response and routing functions (see more in section 5.3.3.3). Jinja2 is a template engine with full Unicode support, which handles HTML templates in a sandboxed environment (see more in section 5.3.4). MarkupSafe is a library that implements a string, or text object handling that allows escaping characters and marks them as “safe” to be used in HTML, avoiding potential malicious code injection. Finally, ItsDangerous is a library that handles data serialization, i.e., it allows to move data in a secure way, no matter how unsecure the environment is. Although user sensitive data, such as passwords, are mostly handled by Werkzeug, it is particularly useful to handle sensitive data in general, e.g., to store user session when using Flask in a cookie, ensuring this token is not compromised.

Flask is probably the framework with easiest setup one can find available. As an example of how to set up an application using this framework, Listing 5.3 shows that with few lines of code it is possible to generate a web page. In this example, after creating an object with the **Flask** package (line 2 in Listing 5.3), by simply adding a function under a route (see more in section 5.3.3.3) (lines 3-6 in Listing 5.3), when running **app** (lines 8-9 in Listing 5.3), the user will be greeted with the message “Welcome to CoreWiki!” when accessing localhost.

```
1. from flask import Flask
2. app = Flask(__name__)
3.
4. @app.route(“/”)
5. def homepage():
6.   return “Welcome to CoreWiki!”
7.
8. if __name__ == “__main__”:
9.   app.run()
```

Listing 5.3 – Snippet of Python code running a simple application using Flask framework.

Ahead, it will be discussed how the developer can add more components to this simple example, increasing the complexity of the platform by adding templates and models.

5.3.3.2. Model-View-Controller design pattern

Regardless of the language used to write the framework, most backend frameworks are set upon the widely used design pattern of "Model-View-Controller" (MVC), including *Subti*Wiki (Zhu & Stülke, 2018; Pedreira et al., 2022).

Introduced first in 1988, the MVC pattern is divided into three interconnected elements, that govern different parts of the logic of the backend – the "Model", and the "Controller", and on the frontend – the "View" (Krasner & Pope, 1988). Together, the different MVC elements divide the logic responsibility of a framework between the server and client (back- and frontend) (Figure 5.4).

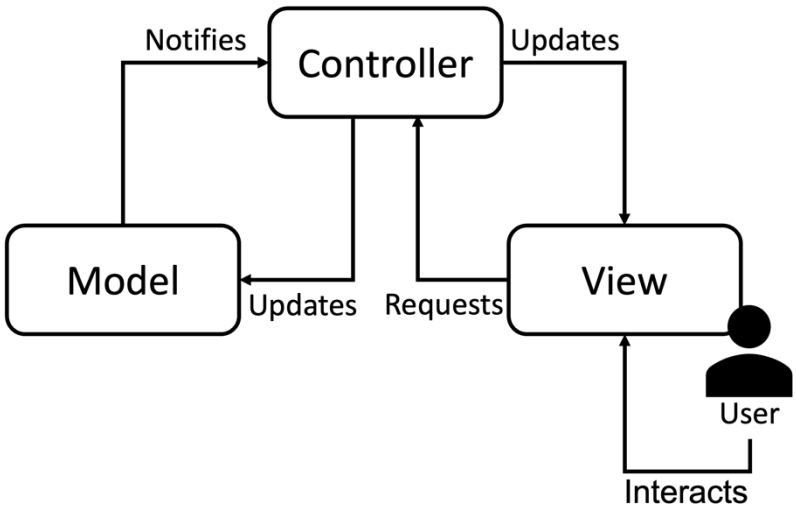


Figure 5.4 – Logic of MVC design pattern. The User interacts with the View to send requests to the Controller. The Controller requests the information from the database and the Model then notifies the Controller with the resulted query. Lastly, the Controller updates the View so that the User can interact with it.

In this design pattern, the Model is the central component and it governs the dynamics associated to the data structure, managing the data directly as well as the logic of the framework. Because of its nature, it is completely independent of the user interface, meaning the user will never have access to this part of the website, but instead interacts with it using the other elements of the pattern. The next component of the pattern is the View and it is responsible for the representation of the data. It is composed of structural components, also called templates, and can be compared to building blocks of a website. This means that there will be different templates for every element of the View, some more static than others. For example, loading a navigation bar that is always the same for every page, and some other building blocks that respond to dynamic information, which will receive the data that comes from the Model. In this element of the pattern, the templates are built with HTML and styling

elements from CSS are loaded accordingly. Templates also allow to run code from JS or other languages, usually responsible for the behaviour of the HTML/CSS or even data elements. The final element of the design pattern is the Controller, which is responsible for mediating the communication between the Model and View. This translates to the Controller handling user requests (create, update or delete information) and manipulating the data to the corresponding model, receiving the information and converting it to the View's building block (Krasner & Pope, 1988).

In practical terms, the MVC pattern also defines how each element interacts with the necessary component of the pattern (Figure 5.4). The user will interact with the View at all times, making use of the Controller when submitting requests to have access or manipulate the data. When receiving the request, the Controller will evaluate it and, if no invalid request has been sent, it will pass the input to the Model. In turn, it will send the response to the View, updating the current visualization the User has access to. A good example of this would be when a user accesses *Subti*Wiki and requests information on a gene, then the following events would happen:

- User accesses View of *Subti*Wiki's main page
- Requests information on *darB* gene
- Request is received by Controller and validates input
- Sends request to Model to extract information on *darB* gene
- *darB* information populates Gene page's building blocks

Notably, there is not one single model for all entities in a database, i.e., each table present in the database will require a distinct and independent model. For example, a model created to handle data on the Gene level will have different attributes and methods when compared to a model created to integrate information on Regulons or Operons. There is an inherent specificity for each data type in the database, requiring a set of unique Model-Controller rules coupled with generic templates in the View element that respond to the data that is being loaded. In *Subti*Wiki's framework, models correspond to PHP classes that handle the data for the specific element. For example, the class Gene, or model Gene, will be responsible for handling the information for a single gene at a time. Accordingly, Controller classes are also specific for each element and usually follow the same model's name, with the addition of "Controller" to it, e.g., "GeneController.php". As mentioned before, only Model and Controller are paired together in terms of specificity, which means that View will have a different handling (see more in section 5.3.4).

5.3.3.3. Endpoint routing

As briefly mentioned before, there is another important aspect in the functionality of every backend that needs mentioning, the **routing**. This is the process of mapping or routing locators requested by the client-side to a certain location. To simply put, this refers to how one user can reach a certain part of the website, or a URL (Uniform Resource Locator) (Chinnici et al., 2007). Usually this is handled by routing functions that map the incoming browser request to the application endpoint. In the MVC design pattern, the Controller class is the one responsible for handling the action and return the response back to the View (see more in section 5.3.3.2). Since the Controller is the main player for handling the process, usually the name of each Controller class will be the one used for the endpoint name. For example, reaching a certain page in *Subti*Wiki requires the framework to interpret the search query from the user and point the View to the correct endpoint. To better understand how it works, Figure 5.5 exemplifies the endpoint for the DnaA Regulon.

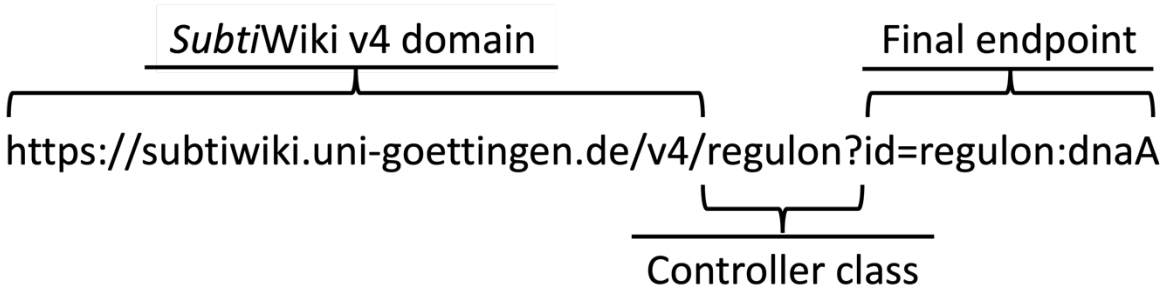


Figure 5.5 – Endpoint for DnaA regulon using current *Subti*Wiki's routing table.

As represented in Figure 5.5, each component of the URL refers to a different element. After the website **domain**, all information is regarding a specific endpoint. Thus, the respective Controller class's name will be the one represented in the URL, which in this case is **regulon**. The question mark, **?**, in the URL is the designator delimiter of the specific endpoint the user tries to reach, or in other words, the result of the search, or query. In this example, the final endpoint will be the entry in the **Regulon** table with an **id** of **regulon:dnaA**. In a general aspect, Controller classes will not only redirect to a specific endpoint, but also how the data will be represented, for example by returning a specific HTML page.

In contrast to the current *Subti*Wiki's framework, where the routing table is set manually, Flask framework handles this process in a simple manner from the developer perspective. Flask relies on Werkzeug routing system to retain the uniqueness of URLs and automatically order the routes based

on their complexity, meaning that no matter how the developer declares the routes, they will always work as intended (Mönnich et al., 2016).

For *Core*Wiki, there are two ways of declaring endpoints and both revolve around declaring functions that will return a specific output, either a value or a page endpoint. For both cases these functions need to be incapsulated in the routing method **route()**, using the proper decorator, **@**. Decorator functions act as wrappers for a certain decorated function, encapsulating its methods and attributes. Furthermore, the **route()** method will always take as an argument a potential URL that will be included in the Flask's routing table. By registering this URL inside the method **route()**, the developer is creating the endpoint of choice. These rules are static for both ways of defining endpoints in *Core*Wiki, however, where they differ is in which object to decorate. While the final object is always the Flask application, it is possible to do it directly (Listing 5.3) or first register a **Blueprint** and decorate to it instead (Listing 5.4) (Mönnich et al., 2016). Blueprints are special Flask constructs that allow to make the application modular, where each module is independent in terms of methods and even routing, in which the developer can use to set multiple subdomains. For example, under *Core*Wiki's default domain, or route, by registering a hypothetical **Blueprint** named "regulon", it will allow to create specific methods for this route space and it is possible to reach it by putting it on the end of the URL: **http://<corewiki_domain>/regulon/**. Under this **Blueprint**, multiple methods that will be specific to this subdomain can be added, for example listing all available regulons, or reaching a specific regulon based on its ID (Listing 5.4).

```
1.  from flask import Blueprint
2.  regulon = Blueprint('regulon', __name__)
3.
4.  @regulon.route('/', methods=('GET',))
5.  def list():
6.     regulon_list = Regulon.query.all()
7.     return render_template('list_of_regulons.jinja2')
8.
9.  @regulon.route('/<id>', methods=('GET',))
10. def view_regulon(id: str):
11.    regulon = Regulon.query.get(id)
12.    return render_template('regulon_page.jinja2')
```

Listing 5.4. – Example of how to set a **Blueprint** and its subdomains using different functions as endpoints with hypothetical regulons from *Subti*Wiki. First is represented the endpoint under the **Blueprint** to list, **list()**, all regulons in the page **list_of_regulons.jinja2**, and second how to represent the endpoint to access the information on a single regulon, **view_regulon()**, in the page **regulon_page.jinja2**.

In Listing 5.2, the **Blueprint** for regulon contains two endpoints, both starting with **@regulon.route()**. For the first case, it is quite simple as the set route is called when entering the subdomain **http://<corewiki_domain>/regulon/**, giving access to the full list of regulons documented in the database. On the second case there is a new parameter being passed to the route, the ID of the regulon in the form of **<id>**. This type of function allows Flask to parse ID's or other parameters that are passed to the endpoint function. In this case, it was defined that the parameter to look for is the ID inside Regulon table. It is important to mention that this can only be done altogether with the models defined previously (see more in section 5.3.2.2), so that Flask knows which entries to fetch. In practical terms this means that the subdomain **http://<corewiki_domain>/regulon/dnaA** will trigger the routing table to forward the request to the function **view_regulon()** while passing "dnaA" as an ID to be looked for in the model **Regulon**.

All blueprints must be registered under the configuration of the application, in this case of *Core*Wiki configuration files. This will activate the internal routing table and Flask will then handle everything after the developer makes use of the necessary method, the **register_blueprint()** function. Only after proper **Blueprint** registration can Flask and Werkzeug activate the routing to the necessary subdomains, e.g., **/regulon** subdomain (Listing 5.5).

```
1. from flask import Flask
2. app = Flask(__name__)
3. app.register_blueprint(regulon, url_prefix='/regulon')
```

Listing 5.5 – Registering regulon **Blueprint** under the main application as a subdomain.

For error handling, blueprints also play a fundamental role by supporting the **errorhandler** decorator, making it easy to build custom error pages. For example, taking the same **Blueprint** for regulons, it possible to add an endpoint to an error page containing details on it (Listing 5.6).

```
1. @regulon.errorhandler(404)
2. def regulon_not_found(e):
3.   return render_template('regulon_error_page.html')
```

Listing 5.6 – Example **Blueprint** to handle error pages.

In this example, the decorator **errorhandler** under the regulon **Blueprint** is used and it creates the endpoint named **regulon_not_found()**, that takes an error from **errorhandler** as argument and returns the page **regulon_error_page.jinja2** when called.

Despite blueprints being defined on the backend, they play a fundamental role in the management of data visualization in the frontend. As can be seen in the previous Listings 5.4 and 5.6, the output of the endpoint functions are return statements of a **render_template()** function. This is a function from Flask that allows to pass certain values from the backend to a template in the frontend. However, instead of redirecting to templates, blueprints also allow the developer to link endpoint pages by using a special function that takes the URL endpoint and the **Blueprint** it belongs to. For example, if it is required to redirect specifically to the list of regulons, this can be done by simply using the function **url_for('regulon.list')**, which takes the **Blueprint** regulon and uses the method **list()** as the endpoint (Listing 5.4). Nonetheless, more regarding this will be further discussed in section 5.3.4.

5.3.4. Structure of the Frontend

5.3.4.1. Jinja2 as a template engine

By default, Flask includes one single template engine: Jinja2 (Grinberg, 2018; Mönnich et al., 2016). However, Flask allows to set up other different engines, but since it is included by default, Flask will set it up and configure it automatically for the developer. This is, as expected, a major advantage when building an application from scratch, as it lifts off of the developer's responsibility to have to set manually a template system.

This template engine is characterized for its ability to enable template inheritance and inclusion, it allows to define and import macros from each template, provides auto escape support, has asynchronous support for generating templates that handle different types of function, allow to write code with similar syntax to Python code, and provide the developer with extensible filters, tests, functions and syntax (Mönnich et al., 2016).

Jinja2 templates are a normal text file that can generate any text-based format, such as HTML and XML. However, this project only focuses on generating HTML code as *Core*Wiki is designed to present only this format. While normal HTML files present the extension **.html**, Jinja2 uses a different extension, **.jinja2**, even though the final result of the file is a readable HTML structure. Within these files, the syntax presented is what can be resembled to HTML, using tags to control the logic of the structure, with the addition that Jinja2 allows to include variables and even expressions. These variables and expressions are treated as placeholders that are ready to receive information from the backend. Upon receiving information, the template will render it into the placeholders.

5.3.4.2. Jinja2 logic control

To make use of variables and expression, Jinja2 offers certain delimiters that must be followed:

- **{{ …… }}** for variables and expressions
- **{% …… %}** for statements
- **{# …… #}** for comments

In Jinja2, variables are passed by the backend to the frontend, or simply by using the **set** method inside two curly brackets. Variables can have many attributes or different elements that the developer can have access to, which can be reached by using a dot, "**.**". For example, if Listing 5.2 gene's object is passed to the frontend under the new name of gene, it is possible to reach its attributes by using **{{ gene }}**, which loads the whole object. To have access to the object's attributes, for example its name, then it is possible to do so by using **{{ gene.name }}**,which upon rendering returns "*geneA*". All attributes defined in the backend that are sent with the object will follow the same rule. Using the same delimiters, it is also possible to use basic expressions, which are commonly used in Python as well (Mönnich et al., 2016). Be it a list, integers and strings, among others, Jinja2 allows to integrate math and logic operations, and even Python methods. Take the same example as before, to capitalize the gene's name after rendering, it is possible to do it by using the method **capitalize()** directly in the Jinja2 delimiter: **{{ gene.name.capitalize() }}**.

As for statements, Jinja2 refers to these as control structures, which control the logic flow, also known as conditionals, loops, macros and blocks. **If** statements and **for** loops are mostly used to build dynamic and responsive pages for *Core*Wiki. Listing 5.7 shows an example of how to use these two elements together, when checking if the passed object contains information, and if it does, then it cycles through it and displays everything in an HTML page.

```
1. {%  if gene  %}
2. <ul>
3. {%  for element in gene  %}
4.   <li>{{  element  }}</li>
5. {%  endfor  %}
6. </ul>
7. {%  else  %}
8.   <p>Sorry, no gene was selected</p>
9. {%  endif  %}
```

Listing 5.7 – Example of Jinja2 use of **if** statement and **for** loops. Based on the logic control, Jinja2 creates an unordered list with the content of the passed object, otherwise a message is rendered instead.

By defining the **if** statement to check if object **gene** exists, Jinja2 creates a logic control and responds accordingly. If it exists, then an unordered list, **<ul>**, is created to display the content of the object, which is iterated by the **for** loop. If the object **gene** does not exist, then a message is rendered instead. This can be further complemented by blocks, which allow to include, based on the logic control, different rendered templates (see more in 5.3.4.3).

Comments can be added to leave out parts of the template that are meant to be tested or simply used to add more documentation on the implemented feature. To comment elements of a template, it is possible to do so by incapsulating said comment in **{# … #}**, for example **{# Adding more information here #}**.

An important feature of Jinja2 is the inclusion of macros, which can be considered equivalent to Python functions. For the same reason as in Python, macros are useful to reuse code meant for the same purpose. For example, creating a special type of formatting or data treatment on the frontend. For this, Jinja2 provides the macro keyword to be used inside a statement delimiter, **{% macro function(args) %}**. Macros can be used in the same scope by simply wrapping the function with the necessary arguments in a variable delimiter, **{{ function(args) }}**. It is also possible to use the same macro within other macros, and for that the keyword **call** must be used, **{% call function(args) %}**.

5.3.4.3. Jinja2 template inheritance and blocks

A way to make templates modular in a Flask application is by breaking them into independent blocks, that can be called and used when necessary. For this, Jinja2 allows to use template inheritance, which will enable the creation of a base skeleton template that contains the core and common elements of the page, then some “blocks” are defined that child templates can use to be rendered on. Normally, a base template is defined that will have the basic HTML skeleton that *Core*Wiki can use. In this file, **base.jinja2**, the blocks are left ready to serve as placeholders that can be rendered when child templates send their own structure to override them. This is usually done by giving a name to the empty block on the main file, which then must be used by the child template that is meant to override the position. An example of **base.jinja2** is shown in Listing 5.8, a simple HTML skeleton containing a **<main>** section within its body tag.

```
1. <!DOCTYPE html>
2. <html lang="en">
3.   <head><meta charset="utf-8"/></head>
4.   <body>
5.     <main>
6.       {% block main %}{% endblock %}
7.     </main>
8.   </body>
9. </html>
```

Listing 5.8 – Example of a template file containing the base skeleton of HTML that can be further extended to children templates. Within this file, a block **main** is defined that will serve as a placeholder for the child template to override.

Inside the **<body>**, a block **main** is defined using the proper syntax, **{% block main %}**, which can be overridden by any child template that inherits the **base.jinja2** file. To inherit this skeleton structure, it is necessary first to extend this file to the child template. Jinja2 also includes the option to expand by using the keyword **{% extends %}** to include in every file that inherits other template files. Taking Listing 5.8 as a base file to extend to a child template, Listing 5.9 exemplifies how this procedure works when creating a **genes_list.jinja2** to load a list of genes. This page will inherit the **base.jinja2** structure and override the block **main** with information from the **Gene** model.

```
1. {% extends 'base.jinja2' %}
2. {% block main %}
3.   <h1>All genes</h1>
4.   <ul>
5.     {% for gene in genes %}
6.       <li>{{ gene.name }}</li>
7.     {% endfor %}
8.   </ul>
9. {% endblock %}
```

Listing 5.9 – Example of **gene_list.jinja2** file. This file extends **base.jinja2** and thus, is considered a child template. Its content is wrapped around the block **main**, which will override the same block on the parent template file. Within the block, a **for** loop is defined to iterate through all genes passed by the backend to the template and list one by one.

In practical terms, what defines a file to be a child template is the use of the keyword **extends**. This ensures that there is a connection between the two files, which is then consumed by the defined block's name. Since this connection is a one-to-one relationship, Jinja2 cannot allow multiple blocks with the same name to be defined simultaneously (Mönnich et al., 2016).

5.3.4.4. Functions render_template() and url_for()

As mentioned before, there are internal ways to redirect to certain templates, either on the backend or frontend. Previous Listing 5.4 in section 5.3.3.3 contains an example of this when defining the View of the **Regulon** model. In this context, when the output of an endpoint function is returning a template, then it is necessary to use the **render_template()** function. This simply tells the framework that when the user is trying to reach a certain view, then the data from that model should be redirected to the selected template, thus the **return render_template('list_of_regulons.jinja2')** is used in line 7 of Listing 5.4. Upon trying to see the list of regulons, the user will be presented with the template file **list_of_regulons.jinja2**.

While the **render_template()** function is from the backend of the framework, the **url_for()** can act as a redirect in the frontend, although not exclusively. However, as mentioned before, this will redirect the user to a specific endpoint function, instead of a template file. Indeed, the endpoint function will return a template by itself however, the required arguments of **url_for()** are the view and the respective function. For example, when adding an anchor tag that serves as a URL for a certain regulon page, by using the function **url_for()** with the regulon page endpoint and the necessary ID of the specific regulon, then Jinja2 will communicate with Flask backend to have access to the required View. The example in Listing 5.10 shows the implementation of how an anchor tag is created containing the **url_for()** function. Inside this function, the regulon **Blueprint**'s method **view_regulon()** is called with the respective regulon **id** (see more in section 5.3.3.3 Listing 5.4). This means that when the user accesses this anchor tag, the web browser will redirect to the regulon with the provided id.

```
1. <a href="{{ url_for('regulon.view_regulon', id=regulon.id) }}">
2.   {{ regulon.name }}
3. </a>
```

Listing 5.10 – Example of using **url_for()** function for the View of a single regulon. **view_regulon()** function was defined in the View of regulon **Blueprint** to require the ID of a regulon as argument, thus all necessary arguments need to be included as well.

5.4. Implementation

5.4.1. *Core*Wiki structure

*Subti*Wiki and CoreWiki are divided in two parts, the frontend and the backend. Here, however, is where the major differences start to appear when comparing both frameworks. While *Subti*Wiki's v4

framework is custom-made, lacks documentation and is built on a LAMP stack, *Core*Wiki makes use of the Flask framework, an open-source, well-known, extensively documented and widely used framework, written in Python, to replace the current structure. For the first time, the full structure of *Core*Wiki is revealed (Figure 5.6). On the backend, *Core*Wiki will still run on a Linux and Apache, adding Flask framework and SQLAlchemy/SQLite on the stack, and the frontend will keep the same triad of HTML, CSS and JavaScript for the website structure, styling and behaviour. There is a note regarding Apache, however. As this software relies on modules that are activated on demand based on the server-side scripting language, it is also important to activate the **mod_wsgi** from Apache that allows Flask to be seen as the framework of use and enable it as a web server (*Apache with Flask*, 2016).

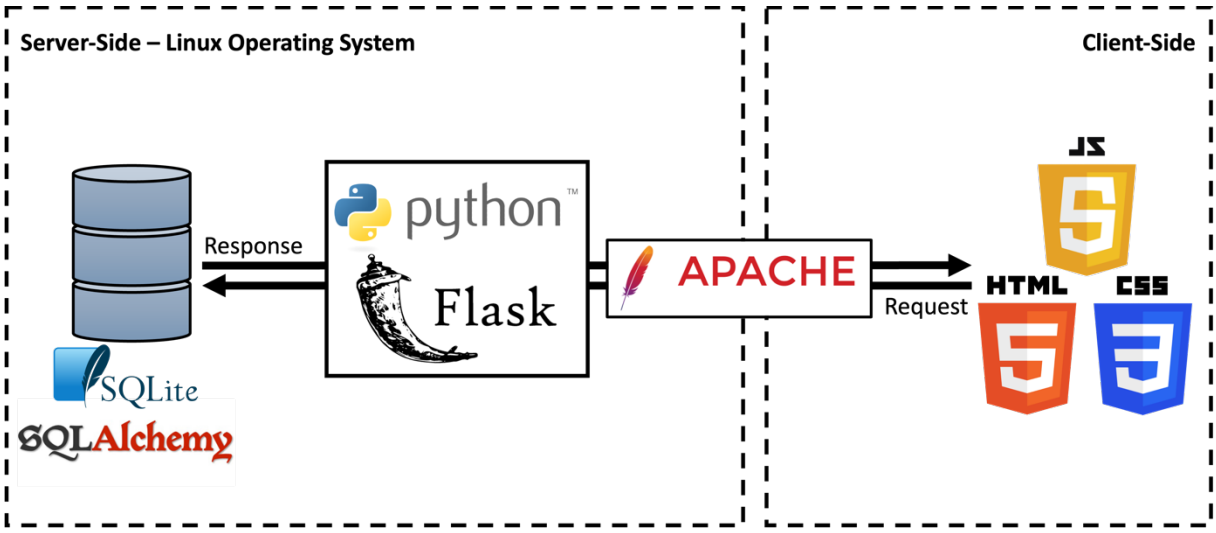


Figure 5.6 – Structure of *Core*Wiki. The server runs on a Linux system, loaded with SQLite as a DBMS and SQLAlchemy as ORM. Flask framework, written in Python, handles the backend logic, mediating the communication between the client-side and server-side. As a web server, Apache handles the HTTP requests between both sides, while the triad HTML, CSS and JS is in charge of the structure, styling and behaviour of the data. The data follows the traditional request/response, handled by Apache and the backend framework (Bayer, 2012; Flanagan, 2011; Hickson et al., 2022; Hipp, 2020; Lie & Bos, 2005; McCool, 1999; van Rossum & Fred L., 2009).

5.4.2. *Core*Wiki architecture

Before going over the Model and Controller creation, it is important to define a few aspects from Flask architecture. *Subti*Wiki's v4 uses the traditional MVC design pattern to handle the communication between the user and the whole stack of technology, but Flask does not enforce this design pattern. Instead, the developer is presented with a "Model-Template-View" architecture (Figure 5.7). However,

Flask gives full freedom to the developer to structure the website the way they seem fit and MVC is a great fit for the structure provided by Flask, which can be easily emulated with a few adjustments.

Although Flask does not require the use of any particular ORM, to build object-oriented data models *Core*Wiki makes use of SQLAlchemy to serve as an abstraction layer between the database and the models, serving as the "Model" in both MTV and MVC. Looking at the "View" class from MTV, by taking full advantage of the routing from Werkzeug, when integrated with data visualization or manipulation methods, Flask emulates the function of a "Controller". On top of this, coupling the routing endpoint with Jinja2 templates enables the final class of the MVC, "View", to be emulated from the "Template" in MTV. And with these considerations, *Core*Wiki can now run an MVC design pattern for its supported information with minimal effort.

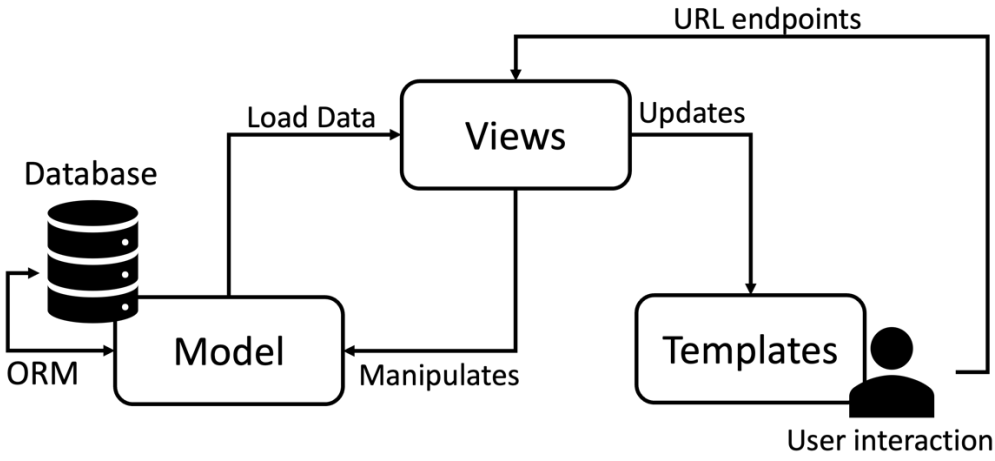


Figure 5.7 – MTV design pattern. The User interacts with the Templates, but requests information by the means of endpoints from the Views. The View, in turn, manipulates the necessary model to retrieve information from the database. The response is then sent to the View, which loads the requested endpoint with the necessary information.

5.4.3. Backend of *Core*Wiki

5.4.3.1. Internal structure

According to Flask's documentation (Grinberg, 2018), it is possible to keep all code in one file. This, however, quickly becomes overwhelming as the application keeps growing larger. To address this, *Core*Wiki's code is organised into multiple modules that act as Python packages and can be imported anywhere whenever necessary. Figure 5.8 shows the project tree of *Core*Wiki.

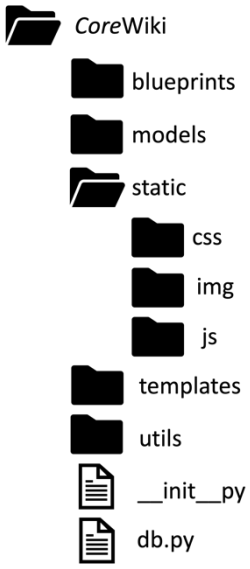


Figure 5.8 – Project tree of *Core*Wiki. Only top-level folders and files are shown.

*Core*Wiki code is divided according to its top-level hierarchy of the MVC model, with **blueprints**, **models** and **utils** directories acting as Python package directories, labelling such folders as importable modules. The **blueprints** directory contains a subdirectory for each corresponding model. For example, the model **Gene** will have a gene **Blueprint** subdirectory with access to an empty **__init__.py** file (to label this folder as an importable module) and a **views.py**, containing endpoint routes as well as other Controller methods. The **model** subdirectory contains all models written for the database. Here, every file will define a table in the database, where is also possible to define some general methods. For example, how search system works is also defined inside the respective model. The **static** directory is the container of JS code, images and styling, while **templates** directory contains all Jinja2 templates. The last importable directory, **utils**, contains Python files that are not considered to be part of models or blueprints, as these files are meant to act as utility functions throughout the framework. An example of this are functions to capitalize all protein names on the frontend, parse JavaScript Object Notation (JSON) format or rendering Markup on the website. Additionally, on the top-level directory of *Core*Wiki a database configuration file was added, **db.py**, where the database is initialized, **db = SQLAlchemy()**. Moreover, an **__init__.py** file was also added with the configuration of *Core*Wiki. This file is of most importance as it is required to initialize the whole application as well as to load the necessary blueprints for the endpoint routing.

5.4.3.2. Database and data models

*Subti*Wiki v4's database is built on MySQL and comprises a large number of tables containing biological data. In a way to conjugate with the abstraction layer provided by the ORM, the development of this framework relied on the use of virtual columns in the database. These virtual columns serve as the communication bridge between the database and the user, which is then handled by the ORM. However, these columns in this **schema** represent a major hurdle as the data is presented in the form of an object, JSON, and most of the documented information is stored in a single entry of this column. This not only poses as a major constraint from the point-of-view of the development, but also from the database maintenance, as it becomes rather complicated to access this information without the proper channels and some of the information is even duplicated in physical columns. Figure 5.9 shows an example of an actual entry, gene *dnaA*, of such virtual column in the live version of *Subti*Wiki v4.


```
{
   "mw":50.695,
   "pI":6.03,
   "Gene":{
      "Coordinates":" 410 → 1,750",
      "Phenotypes of a mutant":[
         "essential [Pubmed|12682299]"
      ]
   },
   "labs":[
      "[wiki|Peter Graumann], Freiburg University, Germany [http://www.biologie.uni-freiburg.de/data/bio2/grau
      "[wiki|Alan Grossman], MIT, Cambridge, MA, USA",
      "[wiki|Heath Murray], Centre for Bacterial Cell Biology, Newcastle, UK [http://www.ncl.ac.uk/camb/staff.
   ],
   "locus":"BSU_00010",
   "strain":168,
   "product":"replication initiation protein",
   "function":"[[category|SW.3.1.1]]",
   "outlinks":{
      "bsu":"BSU_00010",
      "bsupath":"dnaA_410_1750_1",
      "uniprot":"P05648",
      "subtilist":"gene_detail BG10065"
   },
   "regulons":"[[this]]",
   "synonyms":"dnaA",
   "essential":"yes",
   "References":{
      "Reviews":[
         "<pubmed>20157337,21035377,21639790,22575476,22797751,26706151,28075389,30863373</pubmed>"
      ],
```


Figure 5.9 – Portion of gene *dnaA* JSON entry in *Subti*Wiki v4 database. Each key of the JSON represents a major category in the *Subti*Wiki v4 framework (image acquired from JSON Formatter and Validator (JSON Formatter and Validator, 2007)).

JSON format is similar to a Python dictionary, where there is a hierarchy of pairs of keys and their respective values. Inside each value, JSON can admit any structure of data, nested JSON, lists, nested

lists, Booleans, or other simpler structures. In the current *Subti*Wiki v4 format, all keys of the JSON will refer to major sections of the framework. Gene- and protein-specific information, reference to regulon, operons and even literature is all stored in this big entry (Figure 5.9). Although it is very easy to deal with JSON in Python, PHP presents other challenges when manipulating this structure of data. Not only PHP, but also MySQL needs to be compatible to store JSON and the ORM needs to have the necessary methods to decode this format.

Including most of the data of a single gene in a virtual column can also take its toll on memory, as the whole structure needs to be loaded each time, regardless of what to represent. As expected, this quickly becomes a bottleneck in the performance of the server as it requires intensive parsing each time a gene is loaded. In turn, this make the overall load times slower, but it also hinders the search of genes, because JSON elements cannot be indexed by the database.

*Core*Wiki addresses these issues in the database implementation. By relinquishing the JSON column, normalizing the information into multiple tables, and keeping data coherent by establishing relationships, it is ensured that the new database **schema** is fully relational and will be much easier to keep clean and maintain. Not only that, but the loading of very specific elements of each table will make any *Core*Wiki instance to run smoother. Although the main goal is to reduce the amount of JSON in the database, it would be naive to leave it out entirely, as some models may benefit from the flexibility this format can offer. For example, broader information that can be different from element to element could be instead considered to be included in a JSON format, making full use of its flexible properties. It is important to retain, however, that the main elements of the current *Subti*Wiki v4 JSON column, which are usually common across all biological elements, will be treated as independent columns or tables in the new **schema**.

On a general view, *Core*Wiki database structure is as simple as it gets. Designed models are converted into tables that can establish relationships. If necessary, in the case of many-to-many relationships, an additional table, “linker-table”, was added to map the relationships (Figure 5.10A). This is particularly useful as these "linker-tables” allow to map multiple relationships between the same and multiple elements at once with no effort. For example, one gene can be part of multiple functional categories, and a functional category can establish a relationship with many genes, thus many-to-many relationships. This is not necessary for one-to-many relationships as these are a one way only, meaning that the relationship is “unique” for the viewed pair. An example of this is the relationship a gene has with its synonyms, as one gene may have multiple synonyms, but these are specific and unique to that initial gene (Figure 5.10B).

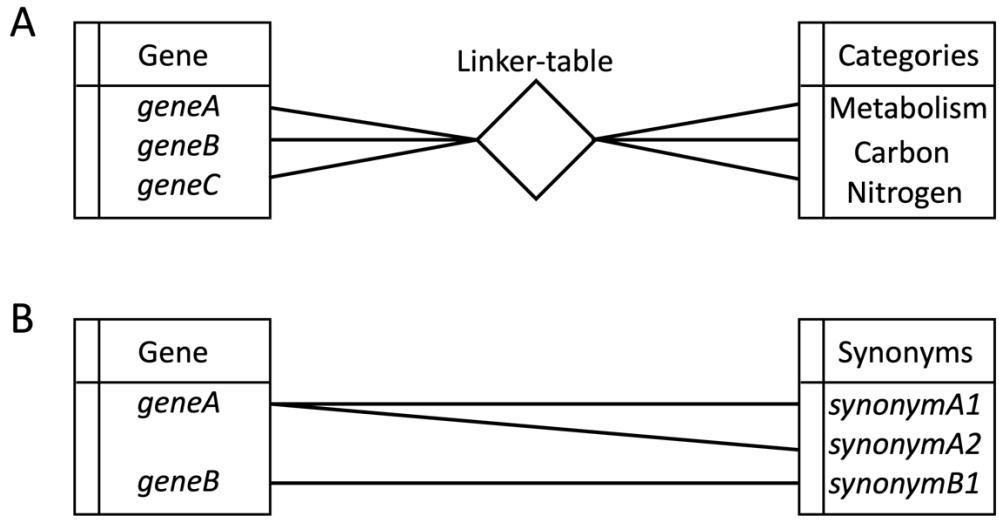


Figure 5.10 – Example of relationships in *Core*Wiki. A – Many-to-many relationship example between genes and functional categories. One gene can establish relationship with many categories, and a category can also establish a relationship with many genes. Having a "linker-table" for this case helps mapping all relationships between the two initial tables. B – One-to-many relationship using gene and its synonyms as example. One gene may have multiple synonyms, but these synonyms are unique to this gene.

Another important aspect of *Core*Wiki is that, together with relationships, all maintenance routines of the database can be easily defined on the creation of the models. This means that when defining a relationship between two tables, there are certain triggers that might be relevant to include. For example, when deleting a gene, all information that is linked to that gene should also be deleted accordingly. With Flask-SQLAlchemy every database trigger and cascading event is easily implemented and most of this is performed behind the scenes by the toolkit. Moreover, this ensures that when migrating the database, or even switching DBMS entirely, these triggers and routines will be maintained.

The creation of data models that are logical and easy to build will support the maintainability of the database and all its structure. For this, *Core*Wiki data models were created based on *Subti*Wiki models, maintaining all the information intact but organised differently. First, it is important to have in mind the exact v4 **Gene** table structure, as well as the full extent of the JSON column structure to successfully create a model that complies with the needs of the data. Figure 5.11 shows *Subti*Wiki v4 **Gene** table structure and the subsequent JSON column structure, **data**.

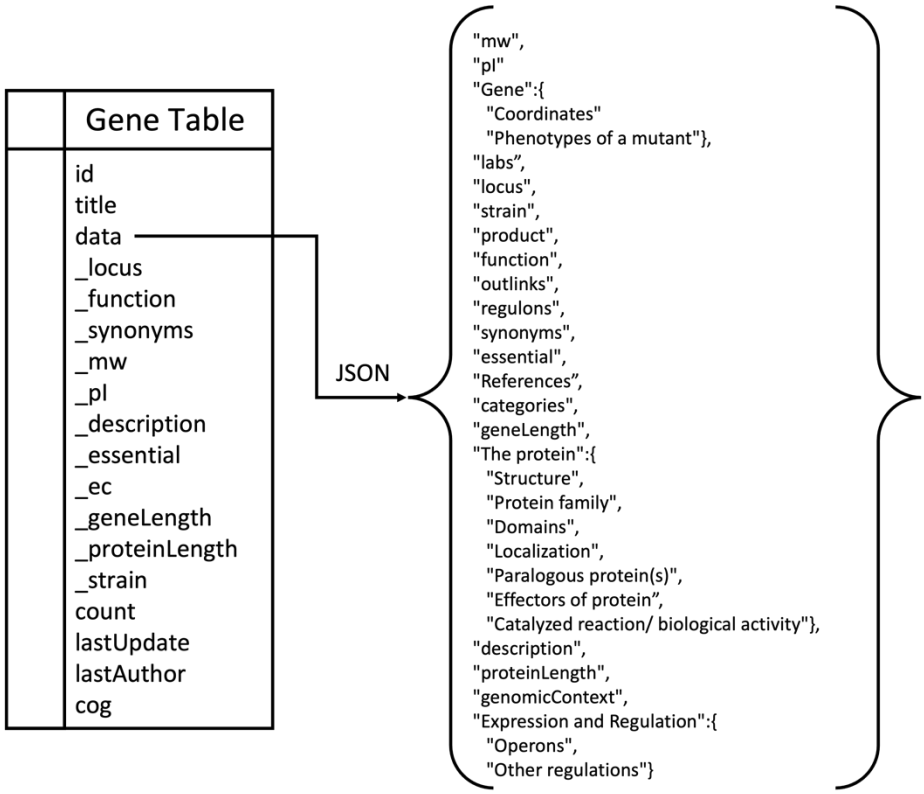


Figure 5.11 – Structure of **Gene** table from *Subti*Wiki v4 **schema**. Also represented all possible elements inside the JSON column, **data**.

As briefly mentioned before, most of the available information is contained inside the JSON, leading to some repetitions. Figure 5.11 shows clear evidence of that, where all columns with exception of **cog**, introduced in Chapter 3, and tracking columns **count**, **lastUpdate** and **lastAuthor**, are present in the **data** column.

The first challenge is to define how to split the data in the old **Gene** model and what is the best way to organise the information in different, yet logical tables. The main goal is to group information based on how specific they are to the respective model. For example, the locus tag, strain, and gene name are only used by the **Gene** entity and there is no need to create additional models for them. Information regarding synonyms, RNA, protein product and genomic context have more complexity and should have their own models with the necessary relationships to the respective **Gene** entity. To better elucidate on this, Figure 5.12 shows exactly which normalization decisions were taken on the previous **Gene** model prior to *Core*Wiki model creation.

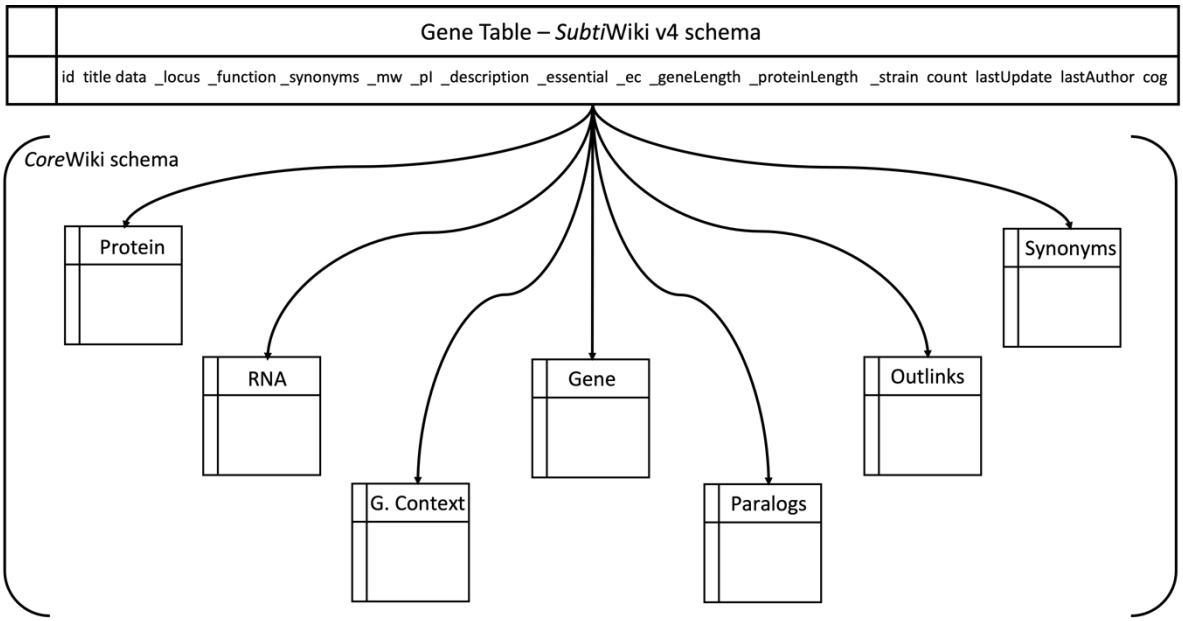


Figure 5.12 – *Core*Wiki database tables containing information regarding **Gene** model from the previous **schema**.

Having determined how tables will be organised, the next step is to create appropriate models. As seen before, Flask allows to create models by defining classes that will contain the specifications of each table. Following the rules described in Listing 5.1 in section 5.3.2.2, Table 5.1 shows which columns and JSON fields from *Subti*Wiki v4 **Gene** table were used to create different models in *Core*Wiki.

Table 5.1 – *Core*Wiki models that were created from the splitting of *Subti*Wiki **Gene** table, and which corresponding fields from the previous table were inherited by the new models.

Model Name	**Inherited fields from *Subti*Wiki v4 Gene table**
Gene	id | title | locus | strain | description | product | essential | ec | labs
RNA	structure | reaction
Protein	mw | pi | structure | domains | reaction | family | localization
Paralogs	paralogous proteins | identity
Genomic Context	gene length | protein length
Outlinks	uniprot | bsubcyc | keg | subtilist | cog
Synonyms	synonyms

Comparably to *Subti*Wiki v4' **schema**, **Gene** table is still the main table of *Core*Wiki, meaning that most of the data and relationships revolve around this table. It is important to mention that SQLAlchemy allows to declare which type of data is accepted in each column, and thus, it is important to set certain rules in each model. For example, the gene **id**, inherited from the previous model, is the unique identifier of each gene. Each gene is labelled with a **sha-1** encrypted 40 character-long string that must be maintained and enforced throughout the database to ensure that relationships exist and are never lost. Although some of the constraints are stricter than others, it is also valuable to allow some fields, such as **labs**, to be flexible. This is thought to be important due to the unpredictable and varying nature of these columns, where they can take up any value in any format. This can be addressed by allowing data structures, such as JSON, to be stored and parsed by Python, which makes the process of loading, accessing and transforming data much easier. It is important to note, however, that using this data structure only for specific columns still allows the database to continue being flexible and efficient. As a result of the choices made to split the initial **Gene** model, the information contained in all columns, JSON data included, have been included in some models and thus, no information was lost. By creating new models for the newly loose data, the information is better catalogued and easier to access while retaining all their inherent relationships with the main **Gene** table.

To address relationships, it important to stress that since most information revolves around the **Gene** entity, there must be the assurance that no annotation is lost in the process of splitting the old model to new tables. When defining the relationships in the models, as shown in Listing 5.1 in section 5.3.2.2, it is equally important to establish additional cascading events that can trigger when gene entries are modified. SQLAlchemy allows to easily define rules in the model by including a new attribute in the relationship: **cascade='all, delete, delete-orphan'**. This attribute allows that when a gene entry gets deleted, for example, then all relationships with the target table are also deleted to avoid having orphan entries. Looking at the given example in table 5.1, if a gene with a relationship with an RNA entry gets deleted, then SQLAlchemy properly removes the respecting entry in the **RNA** table.

Finally, the last implementation of novel database elements are "linker-tables". As explained before these are particularly useful when mapping many-to-many relationships, which is the case of the already mentioned gene-publications relationship, as many genes can share the same publication and one gene can have multiple references. These tables follow the same naming rule: both parent tables' names separated by a "2": **TableA2TableB**, which reads "table A to table B". In the previous example, since the respective tables are **Gene** and **PubMed**, the created linker-table is **Gene2PubMed** (Table 5.2).

Table 5.2 – Structure of **Gene2PubMed** model.

Column Name	Data structure
gene_id	String 40 characters. **Primary key** and **foreign key** to **Gene** table.
pubmed_id	Integer. **Primary key** and **foreign key** to **PubMed** table.
type	String

Notably, the **gene_id** and **pubmed_id** are both **primary keys** and **foreign key** to their respective tables. Defining these attributes ensures that each entry is unique, and by defining a relationship between **Gene-Gene2PubMed** and **PubMed-Gene2PubMed** allows to automatically fetch information on two levels:

- Which publications are annotated to a certain gene?
- Which genes are annotated with a specific publication?

Furthermore, there are also novel models that were implemented in *Core*Wiki that are not included in the current iteration of *Subti*Wiki. For example, there is still some information loaded in the format of MediaWiki, an engine used for previous *Subti*Wiki iterations (Flórez et al., 2009; MediaWiki, 2022). Because of this engine, the information is instead stored in a single table, where each entry represents a single page of its own. These entries, called "Wiki" pages in *Subti*Wiki v4, do not necessarily focus on the **Gene** entity, but rather adjacent or generic thematic, e.g., list of laboratories that work with *B. subtilis* genes, list of constructed plasmids and their details, and even information on various protein families and domains. These pages also count with older visualisation pages, enforcing the necessity to undergo a serious overhaul on both backend and frontend. The entries found here suffered extensive data transformation to reveal which potential columns the model would have. Since these pages require a lot of flexibility, most of these new models must be JSON enabled, or some of the information would either be lost, or would require more models to fit in. With this in mind, the following new models were created:

- Laboratories working with *B. subtilis* – **Labs** model (Table 5.3)
- List of Plasmids – **Plasmids** model (supplementary materials 8.1)
- Protein Complexes – **ProteinComplex** model (supplementary materials 8.2)
- Protein Domains – **ProteinDomains** model (supplementary materials 8.3)
- Protein Families – **ProteinFamilies** model (supplementary materials 8.4)
- Transcription Factors – **TranscriptionFactors** model (supplementary materials 8.5)

As an example, Table 5.3 contains the content of the **Labs** model, while the remaining can be found in supplementary materials.

Table 5.3 – Table structure of *Core*Wiki **Labs** model.

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
name	Varchar	Name of research group leader; Index
email	Text	E-mail of group leader
homepage	Varchar	Homepage of research group
location	Text	Location of research group
pubmed_profile	Text	PubMed profile of group leader
research	Text	Research field of research group
additional_information	Text	JSON enabled

Regarding other novel models, there was also the desire to expand on the interactions plane. As previously commented, the amount of information regarding novel interaction types has been increasing (Link et al., 2013; O'Reilly et al., 2020). Thus, to address this issue yet to be explored by *Subti*Wiki v4, an early implementation of the models of metabolite-protein interaction are first described here. The idea behind it is still to consider a broad and flexible model that allows to establish relationships between elements. These elements are the root of the question, as it is possible to model the interaction of these elements using different approaches. More specifically, the decision to model metabolites interacting with a protein, or if metabolites interact with a particular interaction needs to be made. **Interaction** (supplementary materials 8.6) and **Protein2Interaction** (supplementary materials 8.7) were also created and they store information on the interaction level and map the entries in the **Interaction** model to the **Protein** model, respectively (supplementary materials 8.8). This allows now to decide which model to create for the **Metabolite** entity (Table 5.4).

This new model contemplates a column dedicated for the metabolism it is part of, which gives flexibility for future integration with a potential metabolic pathway. Targeting the interaction with proteins, although metabolites may interact with proteins, for the *Core*Wiki concept it is easier to target the interaction they are part of instead. The **Metabolite2Interaction** model maps which metabolite targets which interaction (supplementary materials 8.9). This enables the framework to automatically include the effect of the interaction in the model, making it easier for representation. For example, it has been seen recently that the proteins DarB and Rel interact, but when the metabolite cyclic-di-AMP, c-di-AMP, is present, it binds to DarB relieving the interaction with Rel (Krüger et al., 2021).

Table 5.4 – Metabolite model in *Core*Wiki.

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
name	String	Name of metabolite
metabolism	String	E-mail of group leader
pubmed	String	List of PubMed ID's associated to metabolite
additional_information	Text	JSON enabled

Although c-di-AMP interacts with DarB, from the development perspective it is easier to enable the effect of this interaction through the interaction itself and so, the decision to establish a relationship between **Metabolite** and **Interaction** models was made (Figure 5.13A). Moreover, this also allows to make full use of the upstream information of the **Interaction** model, meaning that through relationships it can reach the proteins mapped in the model, facilitating the representation of the interaction in the frontend (Figure 5.13B)

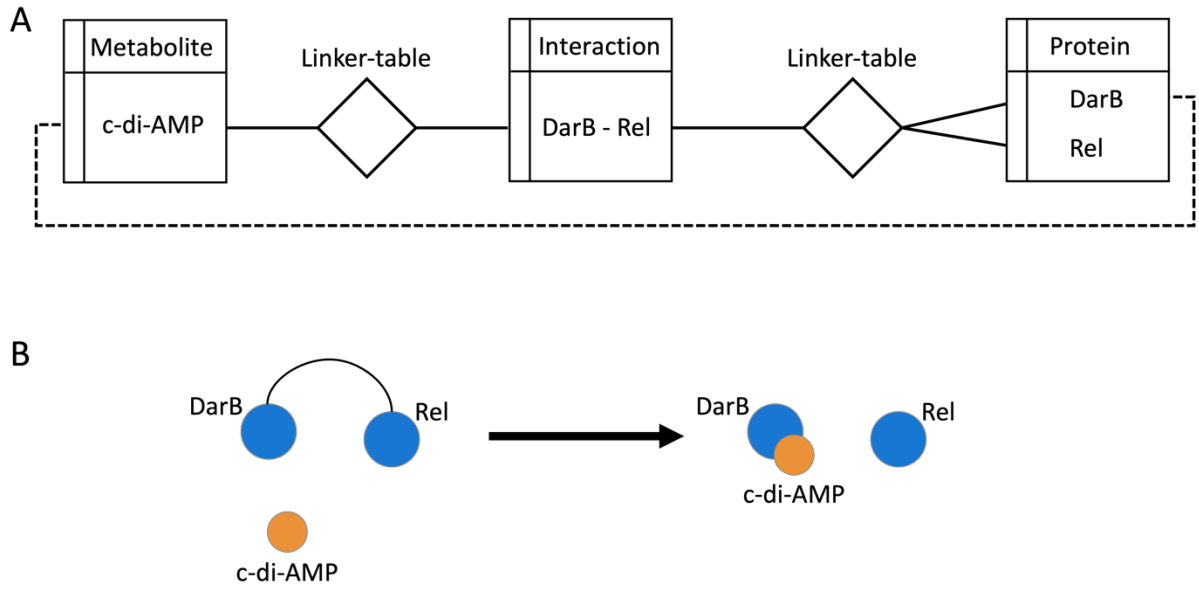


Figure 5.13 – A – Representation of the **Metabolite-Interaction** model and relationship with the respective linker-tables and with the **Protein** model. Solid lines (──) represent direct relationship, and dashed lines (----) represent indirect relationship. B – Future potential representation of **Interaction** model in *Core*Wiki. DarB-Rel interaction represented, when c-di-AMP interacts with DarB, the connection with Rel is lost.

5.4.3.3. Controller classes and endpoint routing

As mentioned previously, controller classes allow to manipulate how the end user will access and manipulate the information contained in the database. In Flask framework, these also serve as the endpoint routing, meaning which page the user accesses to visualize the data needs to be defined here. As explained before, these rules are all set under the **blueprints** directory, where each endpoint subdomain will have a **views.py** file, describing functions that will act as Controllers. Importantly, it is necessary to understand that although the Controller interacts with the Model, not every model will have a specific **Blueprint**. This is due to the fact that endpoints are merely a subdomain, containing methods to access and manipulate the data, regardless of which table it comes from. Instead, defining subdomains that will then take full advantage of the relational model and extract all information from a specific entry will make the representation of the data possible. In the current implementation of *Core*Wiki, Table 5.5 shows the list of all blueprints and which routes/subdomains were implemented to accommodate the data.

Table 5.5 – List of blueprints and the respective subdomains implemented in *Core*Wiki.

Blueprint name	Route	Comment
gene	'/gene/'	Access gene-related pages
category	'/category/'	Pages with functional categories
home	'/home/'	Homepage of *Core*Wiki
homology	'/homology/'	Result of Protein Homology analysis
operon	'/operon/'	Information on operons
regulon	'/regulon/'	Information on regulons
pubmed	'/pubmed/'	API and view of PubMed entries
wiki	'/wiki/'	Contains subdomains for old Wiki pages

Each **Blueprint** has access to a specific subdomain name that is found under "Route". As explained in 5.3.3.3, to access the subdomains it is required to add the necessary route to *Core*Wiki's domain, e.g., **corewiki_domain/gene/** to access all methods inside **gene Blueprint**. Within each **Blueprint** it is possible to define multiple Controller-specific methods to access different pages with specific views. As an example, Table 5.6 contains the methods developed for the **gene Blueprint** as well as which attributes each endpoint requires.

Table 5.6 – List of endpoints and methods implemented for the **gene Blueprint** in *Core*Wiki. All routes included are subdomains of the '/gene/' subdomain.

Endpoint	Route	Return
list()	'/'	"page_gene_list.jinja2" with all genes
view(id)	'/<id>'	"page_gene_view.jinja2" for gene with id "*<id>*"
search()	'/search'	"page_gene_search.jinja2" with query results **view()** endpoint with specific gene
edit(id)	'/edit/<id>'	"page_gene_editor.jinja2" for gene with id "<id>"
random()	'/random'	"page_gene_view.jinja2" for random gene with "*<id>*"

The endpoints implemented for this **Blueprint** require tight communication with the model. For instance, to access the list of all genes, using **list()** endpoint, the result of **Gene.query.options(load_only('id', 'name')).all()** is passed to the template **page_gene_list.jinja2**. For the endpoints with a necessary argument, such as an **id**, it is passed by the frontend to be used in a query by the methods **view()** and **edit()**. The **search()** method is implemented in a way that if the search criteria was too broad, it returns a list of potential matches based on the query. If the search criteria is a match with the database query, then it passes the **id** of that gene to the **view()** endpoint. The **random()** endpoint just loads a random gene from the database and passes its **id** to the **view()** endpoint as well. The **edit()** endpoint is the one that stands out as it uses a series of methods from a **form.py** that is contained in the same **Blueprint** directory. Although not yet implemented, Flask uses web forms to handle editors by the use of the package FlaskWTF-FlaskForm, which serves as a scaffold for editors. With the exception of gene **Blueprint** (Table 5.6), which requires more endpoints since it is the main entity, all other models presented here are only implemented with **list()**, **view()** and **edit()** blueprints.

5.4.4. Frontend of *Core*Wiki

The main objective of this work focuses on the development of a novel framework ready to include not only *Subti*Wiki v4's but any prokaryote's information. Coupled with the fact that some of the frontend of *Core*Wiki was developed within the scope of a Master's Degree research project, this part of *Core*Wiki is not extensively evaluated here. Instead, generic aspects of the conceptualization and some implementations are shown. Regarding the design and styling choices, these were kept from *Subti*Wiki v4, namely colour scheme, overall structure and design appearance. As mentioned before, Jinja2 templates were used to accommodate the information in a refreshing way when compared to the current templating system from *Subti*Wiki v4. Regardless of the improvements, *Core*Wiki counts

with the same structural elements as *Subti*Wiki v4, more specifically HTML and CSS. The CSS, however, has been improved slightly by using of Syntactically Awesome Style Sheets (Sass) (*Sass - CSS with Superpowers*, 2022), which is a pre-processor scripting language that is compiled by the backend into CSS upon rendering a webpage. An advantage of using this is to define variables that can be used globally, for example the colours of the website, **$primary-theme-colour: #1976d2**.

5.4.4.1. Template structure

Flask uses Jinja2 extension to make use of a template system. There, it is possible to find scaffolds of HTML code ready to render information passed by the backend. Indeed, the communication between backend and frontend is tightly connected in the Flask framework. Since there are many endpoints, the implementation of the templates must be thought thoroughly. Here, organisation plays a fundamental role as multiple templates were created to render different parts of each page (Table 5.7).

Table 5.7 – Organisation scheme of templates in *Core*Wiki. **{name}** refers to any page element of *Core*Wiki.

Name	Level	Comment
base_{name}.jinja2	Main Structure	Main HTML structure with empty blocks
page_{name}.jinja2	Subsection	Extends from base templates
_{name}.jinja2	Complimentary	End complimentary section of subsections

All template files are stored under the **templates** directory and follow the same naming rules. Templates that act as a base of other templates will be named with the prefix of **base_**, for example the template to render data was named **base_data.jinja2**. The files with the prefix **page_** can also be considered the main structure for other building blocks, however these templates still extend from previous **base_** templates, e.g., **page_gene_view.jinja2**, which is responsible for loading a gene's page, extending from **base_data.jinja2**. Finally, pages that start with **_** followed by a name of a page, are considered the final complimentary part of a **page_** template and are manually included in the previous template file. For example, the file **_gene_table.jinja2** is manually included in the **page_gene_view.jinja2** to load a table with some generic aspects of the gene.

5.4.4.2. *Core*Wiki pages

*Core*Wiki uses the *Subti*Wiki design and data to test both backend and frontend and thus, here the implementation of *Core*Wiki's frontend is applied to *B. subtilis* and *Subti*Wiki's design. The idea is to

recreate *Subti*Wiki's pages using Jinja2 templates as much as possible while improving a few aspects and adding some extra features still unavailable in v4. As expected, there are still some pages left to be implemented.

*Core*Wiki's homepage (Figure 5.14) is very similar to the one found in *Subti*Wiki v4, without the buttons for other features, such as the genomic browser. Although this is due to the lack of implementation of such features, an important change made so far is the removal of the second search button, which was meant to search on a broader way. In *Core*Wiki, the search for genes was designed to include broader elements, such as gene's name, locus, function and even description and thus, condensed every search pattern in a single hit of a button.

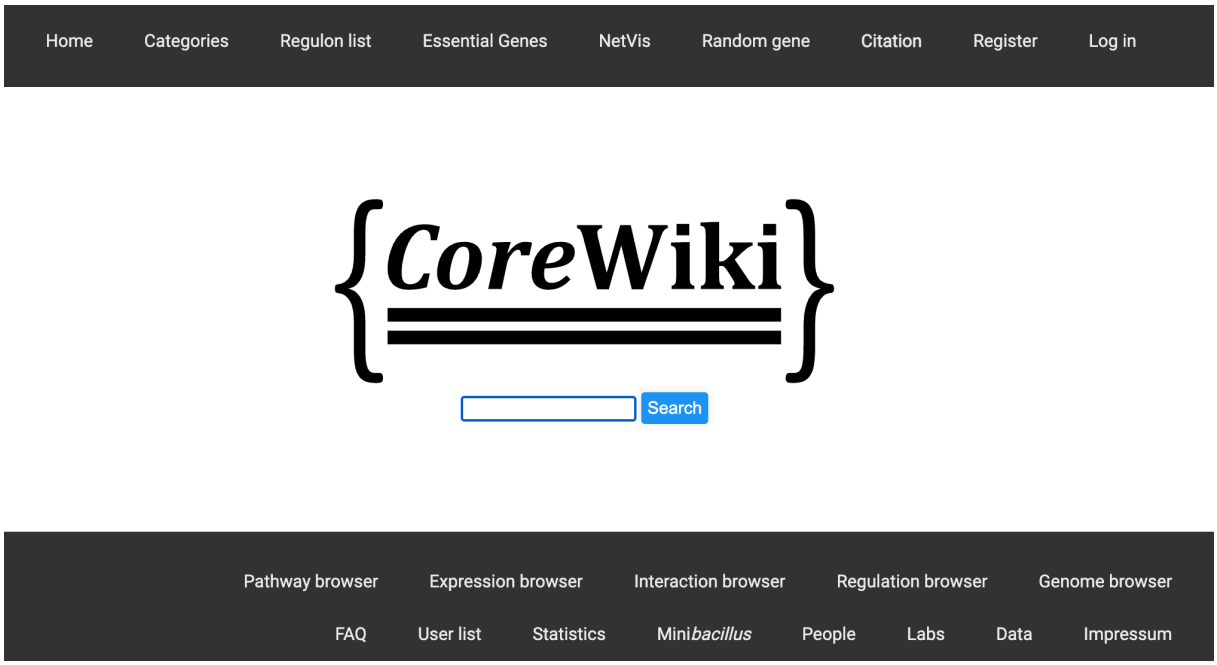


Figure 5.14 – Homepage of *Core*Wiki. Search bar changed to include only one search button with increased functionality.

Each page in *Core*Wiki is expected to render the data that can be seen in *Subti*Wiki v4, relying on the same structure to convey information to the user. Information on genomic aspects, protein details, functional categories, protein homology, literature and regulatory elements are just a few of the many sections from *Subti*Wiki v4 that need to be included in this framework. Importantly, although *Subti*-Wiki's data and design are being used, the main idea of making this platform available for other organisms and other types of data, even if from *B. subtilis*, is retained.

Taking the models and applying them into a frontend can sometimes be challenging. An example of this is the PubMed section in the current *Subti*Wiki v4, which fetches the respective entries from the database in the format of HTML code, meaning that every entry in this table is saved in HTML format. PubMed API (National Center For Biotechnology Information, 2010) sends the requested information, which is then stored directly in the database with the frontend elements. From the frontend perspective, this is convenient as it is possible to simply load the HTML directly without having to manipulate the data to fit a certain template. However, to edit this information is rather troublesome. *Core*Wiki still uses the same API to fetch data, but ensures the data is transformed properly for storage, meaning it is never saved as HTML. The final result should still be the same as *Subti*Wiki, as the same styling is to be preserved. This case is a good example of how changing a model but retaining its final frontend implementation can be challenging.

One of the newest additions that is yet to be implemented in *Subti*Wiki v4 frontend, is the "Wiki" pages. As mentioned before, these pages are still part of the MediaWiki structure that have been used since version 1 of *Subti*Wiki (Flórez et al., 2009; MediaWiki, 2022). With the newly developed models (see more in section 5.4.3.2), new Jinja2 templates to host this data were developed and it is now possible to find dedicated pages under the **/wiki/** subdomain. As an example of how the information is displayed in the old format, Figure 5.15 shows the comparison between the implementation of the old format (Figure 5.15A) and the *Core*Wiki approach (Figure 5.15B). For the latter, a friendly search bar was included that allows users to search for a specific laboratory without having to scroll through a list of names.

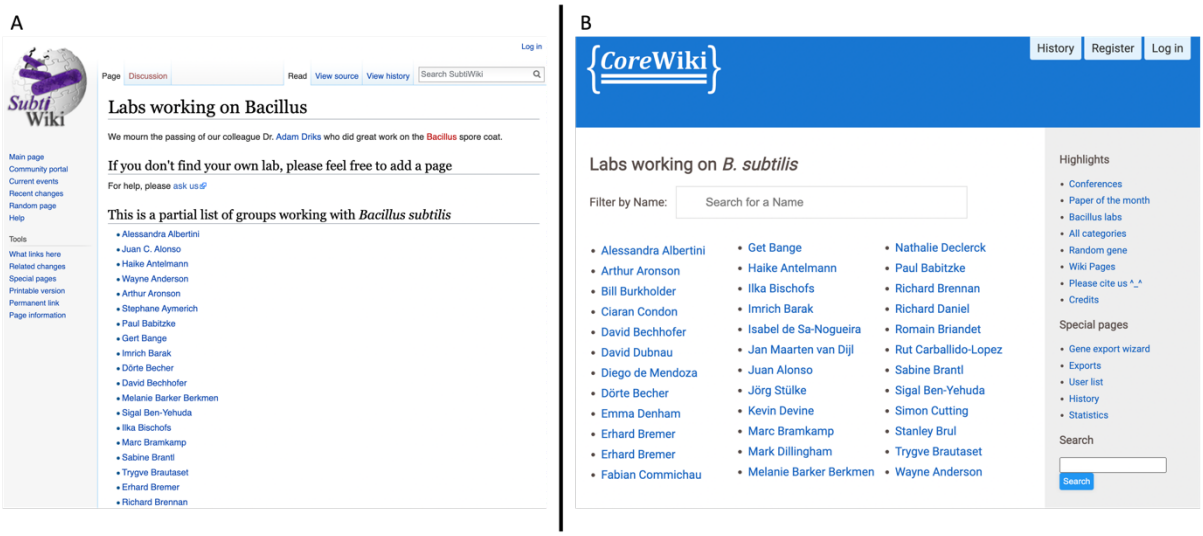


Figure 5.15 – Difference of implementation of the Wiki **Labs** page in different frameworks. A – List of Labs working on *B. subtilis* in the current MediaWiki implementation. B – Labs working on *B. subtilis* in the

*Core*Wiki framework with reworked models. A filter bar is at the top for easier searching among group leaders.

Upon loading a Labs' page, the view is also clearly different from the original (Figure 5.16A), but it keeps the new and refreshing style from *Subti*Wiki v4 (Figure 5.16B). All the remaining new models follow this same structure and design as the one shown for **Labs**.

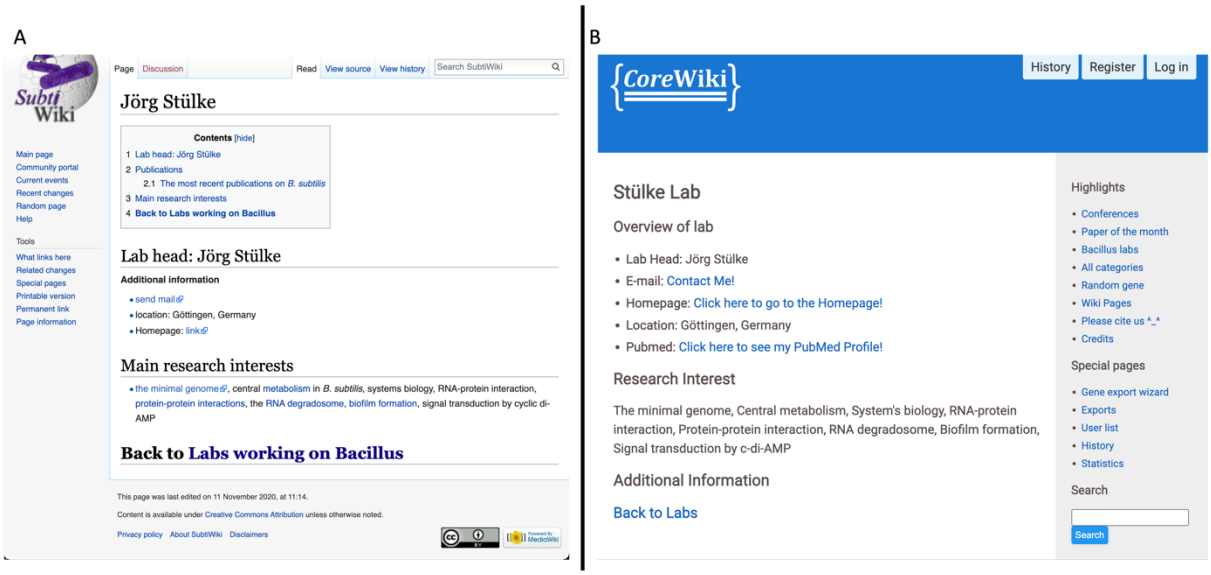


Figure 5.16 – Implementation of the Wiki **Labs** view page for Jörg Stülke's research group in different frameworks. A – Current MediaWiki implementation. B – *Core*Wiki implementation.

Finally, in *Subti*Wiki v4 the user has access to a list of Markup language functions to enable internal or external redirect of information. The theory behind this language is simple, a parser reads the text and internally creates a URL on the frontend that redirects the user to the indicated handled page. For example, in the current v4 implementation, if the user would like to create a link to the *darB* gene's page in the page of *rel*, then it is possible to do so by adding the following Markup in the gene editor of *rel*: **[gene|darB]**. By reading this, the parser replaces the Markup on the frontend page of *rel* gene by automatically creating an anchor tag with the URL to *darB*'s gene page. From the storage perspective, it is much easier to have a short Markup, such as the given example, than storing a lengthy HTML anchor tag. For this, all Markup follow a certain rule concerning their use: **[handle_name|element_to_link]**. Using the previous case, **gene** was the **handle_name** and **darB** was the **element_to_link**. Internally, the handles will usually link to a certain endpoint from a **Blueprint**, however this can also only label certain elements of the page to be subject of a special formatting which is the

case of PubMed formatting. The Markup included in the previous version were kept, but new Markup had to be developed for the novel models (Table 5.8).

Table 5.8. – Markup in *Core*Wiki for the new **Wiki** models.

Handle	Markup	Comment
wikitranscriptionfactor	[wikitranscriptionfactor|]	Links to transcription factor page
wikidomains	[wikidomains|]	Links to protein domain page
wikiplasmids	[wikiplasmids|]	Links to plasmid page
wikilabs	[wikilabs|]	Links to lab page

5.5. Conclusion

Scientific fields are now facing the great expansion of technology, which allows research groups to increase their data generation like never before. To address this explosion of information, biological databases play a fundamental role in the organisation of information. *Subti*Wiki is the popular database for the model organism *B. subtilis*, counting with millions of requests yearly and playing a crucial role in supporting scientists that work with this organism. To be able to keep up with the increasing data, *Subti*Wiki must be ready to support the development of new features and for this, maintainability plays the utmost important task. This platform has started as a MediaWiki application and recently, in its fourth version, a dedicated framework was created to support its development. However, this iteration of *Subti*Wiki presents major challenges in its maintainability as it lacks proper documentation to assist the developer in the process of maintenance and active development. Moreover, although it is working as intended, some implementations on the backend are old-fashioned when compared to modern methods implemented by open-source frameworks.

To address these flaws, *Core*Wiki was created and aims to serve the scientific community by providing a fresh and modern development of databases for any prokaryote. To test the usability of this new framework, it targets first to replace the current implementation of *Subti*Wiki, using its data to address its fundamental challenges in maintainability while keeping the design as close to the original platform as possible. *Core*Wiki counts with an improved relational database schema, defined with SQLite and SQLAlchemy, with new models that make the information more accessible than ever, while keeping intact every relationship all biological elements can establish. Moving away from storing all information under a JSON virtual column will make *Core*Wiki models more efficient and easier to maintain, not requiring extensive handling of such data structures. Relationships allow to integrate routines

for maintenance and implementing triggers to activate cascading events can be easily accomplished in the definition of each model, which will ensure that once information of an entity is changed, every dependency will react accordingly.

*Core*Wiki relies on the popular Flask framework, a platform written in Python that is widely used by major companies and organisations. Flask is considered to be a minimal framework, reducing the codebase by a significant portion while doing all the heavy lifting of routing and database logic. This frees the developer from the burden of having to manually implement every aspect of a website, making more time for the implementation of novel features to integrate new types of data. The notorious less complexity of the platform, providing the necessary documentation, was the major achievement when tackling the current *Subti*Wiki framework's issues. Keeping the same MVC design pattern, everything about *Core*Wiki is simpler than *Subti*Wiki, where Models, Controllers and even Views are working together in a clear and intuitive manner.

This new platform also contemplates the expansion of potential new information. More specifically, the lack of models for the old "Wiki" pages of *Subti*Wiki v4 was addressed. Generic information that was once put together in a single database entry, is now divided into multiple tables with a supporting Model and Controller that handle the data manipulation. Data regarding protein domains, transcription factors, plasmids, laboratories that work with *B. subtilis*, protein complexes and list of biological materials are now added to the remaining models. Coupling these with the new Jinja2 template system allows to better represent not only new models, but also the old ones with no effort for the developer. Additionally, a new metabolite-protein interaction was introduced, opening up the possibility to include information on this level of interaction for future data integration. This is of particular interest as more data regarding non-protein-protein interaction emerges. This new model opens up a new precedent on the unexplored frontline of novel data implementation and representation in *Core*Wiki.

More flexibility, maintainability and development support are only one of the many improvements *Core*Wiki includes that will help reach new heights in science. This platform aims not only to include information on one organism, but to serve as the scaffold for any prokaryote, meaning that even the other developed Wikis, such as *Syn*Wiki, will be able to benefit from this framework. Its modularity and flexibility allow to integrate different levels of information and thus, organisms such as JCVI-syn3.A that have substantially less information than *B. subtilis*, will not see its information integration hindered at any point. There is, however, room for improvement, as many features from *Subti*Wiki's frontend are yet to be implemented. For example, the Genomic Browser and Interaction Browsers are a hallmark in the viewership and popularity of *Subti*Wiki and must be implemented in a near future. It

is also expected that once features like the Interaction Browser are implemented, the new metabolite-interaction model will fully benefit from it, adding even more value to this platform.

Finally, *Core*Wiki aims not only to be used in our research group but also to expand and be used by other scientists. Making *Core*Wiki open source and using it to its full extent its simple implementation will facilitate the integration of other organisms. Either publishing online or simply run locally just for the group members, research groups will have every reason to implement *Core*Wiki to aid in their research and help scientists postulate new hypotheses.

Chapter 6 – Discussion and outlook

6.1. The current state of biological databases

*Subti*Wiki is the biological database for the model organism *B. subtilis* and it has been increasing in popularity ever since its first implementation (Flórez et al., 2009; Zhu & Stülke, 2018). With the rise of such technology to catalogue, organise and annotate biological elements, emerging data finds its way into the hands of scientists in a faster way. However, building these structures is often seen as a big challenge due to the complexity and variety of biological data (Agarwala et al., 2016; Alkan et al., 2011; Loman et al., 2012; Manzoni et al., 2018; Mardis, 2017; Reuter et al., 2015). While this is particularly true for model organisms, that are extensively studied and thus, have increased data, organisms with less information have other limitations tied to the lack of data. All databases should address these issues, but should be built around the same principles regarding biological data:

- Biological data has increasing complexity, for example data from interaction data or metabolic pathway show high degree of complexity to integrate
- Biological information has high variety, for example information regarding genes, proteins, and other elements
- Biological data is fast paced, especially in the current technology era with access to high throughput techniques that generate extensive amounts of information

All databases take these principles at hand and attempt to integrate readily available information while providing the necessary biological context supporting it. It is important to stress, however, that without biological context, all data is rendered useless and thus, biological databases continuously strive to provide with curated information. A way that biological databases have addressed these points is by adapting the relational model (Codd, 1970), which allows to preserve the complex intrinsic relationships between biological elements.

6.2. *Subti*Wiki and Model Organism Databases

As discussed in Chapter 2.2.1, there are multiple model organism databases that have a record of popularity among the scientific community, and *Subti*Wiki is the perfect example of one of these successful databases. In its fourth version, this platform has a good history of increasing amount of information stored in its database (Zhu & Stülke, 2018), and here it was shown that this trend is still maintained (Pedreira et al., 2022). With novel features, *Subti*Wiki continues to grow in viewership and according

to the access statistics, it counts with an average of 5600 users per month and a solid viewership across the year (Figure 6.1).

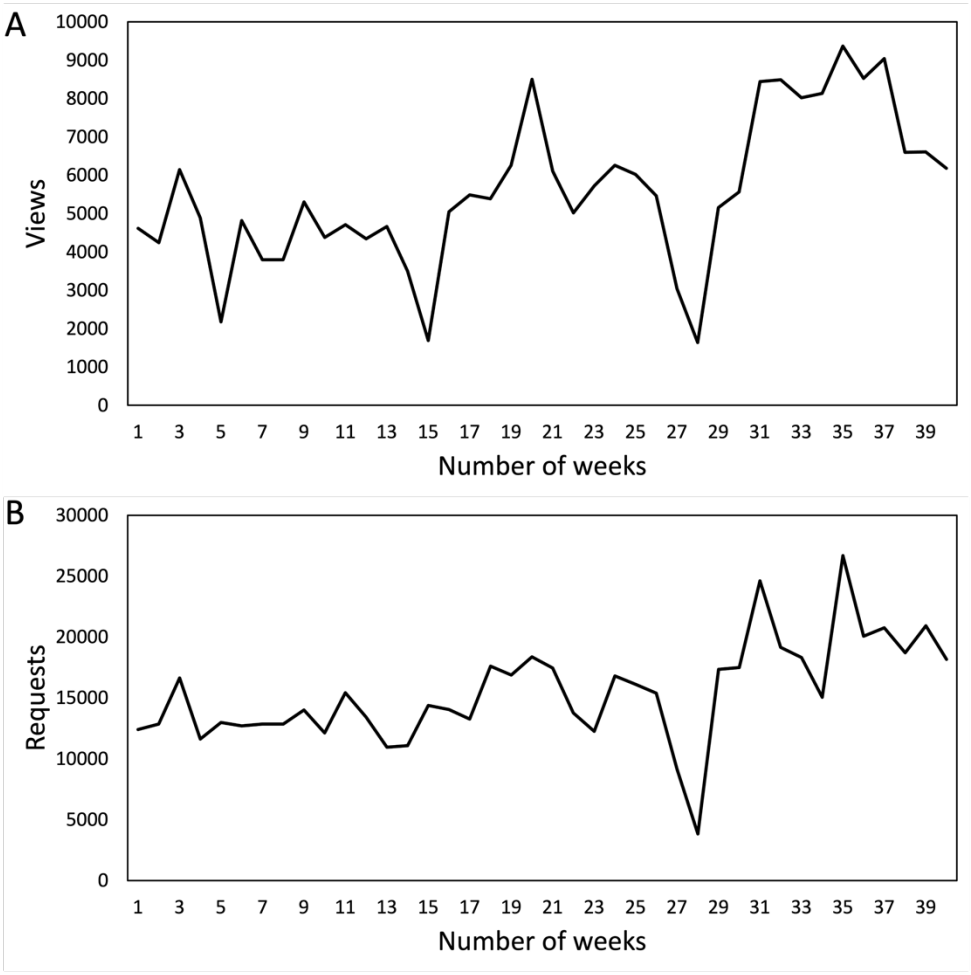


Figure 6.1 – *Subti*Wiki statistics of access. Data corresponds to the period between 1st of July of 2021 and 30th of March of 2022. A – Number of weekly views. B – Number of weekly requests.

In the timespan displayed in Figure 6.1, *Subti*Wiki had over 244000 visits, referring to independent and unique user accesses to *Subti*Wiki, and 670000 requests, representing every query made to the database, reinforcing the position that *Subti*Wiki plays in the daily life of researchers. This is the result of the major effort that was put together to include high level curated data. With more information than ever before, and still growing, the unknown part of *B. subtilis* is slowly being unveiled and *Subti*Wiki serves the community by putting all the information in context and under intuitive visualization.

Most biological databases strive to provide more and better information. Since most microorganisms share the same base biological elements, i.e., genes with a sequence, genomic contextualization, a function, etc., where these platforms struggle to stand out is on the quality of the data and how

to present it. Although most community databases have a big team behind their databases to handle curation and software development, e.g., EcoCyc (Keseler et al., 2021) and Saccharomyces Genome Database (Cherry et al., 2012), *Subti*Wiki counts with a much smaller team to provide the same quality of a database. Regardless of this fact, *Subti*Wiki quality is widely acknowledged by the scientific community and thus, is now considered the preferred platform among *B. subtilis* scientists. Among the discussed MODs in Chapter 2.2.1, despite its small team, *Subti*Wiki provides one of the best top-level data representations, with a multi-level data integration in intuitive biological networks. A refreshing and modern design attracts scientists to use this platform as most of the times the looks of a website, regardless of its data, can pose as serious barriers when attracting new users. An example of this, is the very popular EcoCyc database, which besides its popularity, still shows major differences in the design, especially when compared to its counterpart in *Subti*Wiki (Figure 6.2).

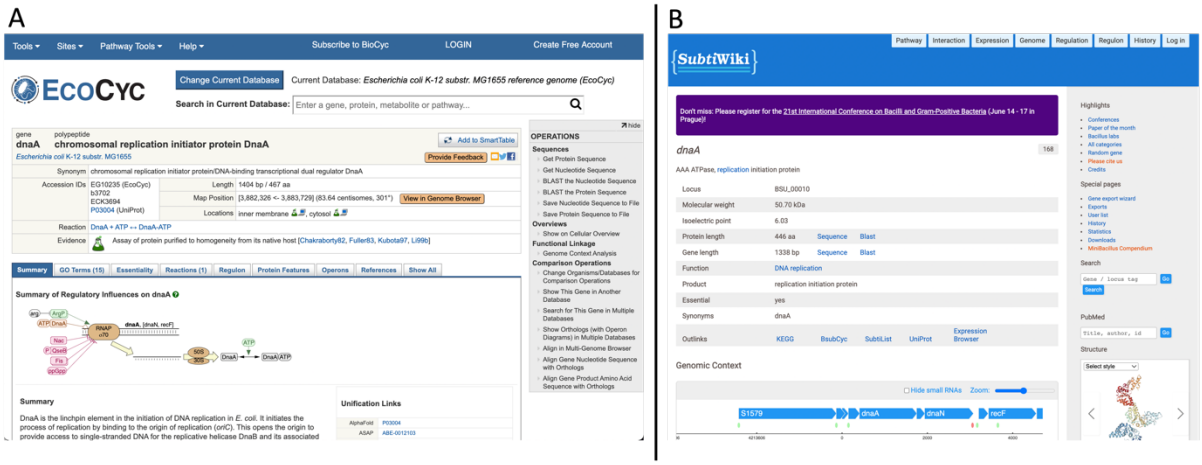


Figure 6.2 – Comparison of the design of two popular databases for the same biological element, the gene dnaA. A – *dnaA* gene page in the EcoCyc database. B – *dnaA* gene page in *Subti*Wiki v4.

Indeed, how the information is presented plays a fundamental role in the integration of data, allowing to have a clearer view over the desired data. However, this can often be seen as a struggle by some databases as the technology to implement novel ways of presenting emerging information can be hard to use and develop.

6.3. Current data in *Subti*Wiki

Addressing the unknown proteins of this organism, the protein homology integration is now one of the most popular sections of the platform. Coupling this with the COG database (Galperin et al., 2021), *Subti*Wiki elevates itself as a biological database and, more importantly, steps out of the shadow of

the data gathering only function and for the first time does not rely on the results from research groups to increase its contents. The incorporation of a new protein homology analysis set up a new precedent for *Subti*Wiki on the contribution of the available annotation of *B. subtilis* (Pedreira et al., 2022). Moreover, the value of *Subti*Wiki keeps increasing with the addition of state-of-the-art information, such as the MiniBacillus Compendium (Michalik et al., 2021), novel expression datasets (COPR library) (Senges et al., 2021), new detoxification of toxic metabolites categories. Furthermore, *Subti*Wiki has moved away from the long-term essentiality binary by introducing a new category, the quasi-essentiality. Although anyone can compile these data and integrate it directly in a personalized platform, research groups trust *Subti*Wiki and maintain a close cooperation with the developers. Thus, *Subti*Wiki gains advantage over any other database by being the first to implement emerging data, e.g., MiniBacillus Compendium and COPR library.

The development of new methods to implement newly available information have the greatest advantage of being able to be transplanted to the remaining Wikis developed by the inhouse research group. An example of this is the homology analysis that can be easily expanded to different organisms, as the only requirement is to have a library of proteomes. With other developed databases for different organisms, it was shown that using the same approach can provide a valuable enrichment of the annotation for less studied organisms, such as the JCVI-syn3.A.

6.4. Open possibilities by expanding *Subti*Wiki framework to JCVI-syn3A – *Syn*Wiki

*Syn*Wiki was originated in response to the creation of the first synthetic microorganism with a minimal genome, *M. mycoides* JCVI-syn3.A (Hutchison et al., 2016). In order to catalogue all the known information, the current *Subti*Wiki framework was used as the host of this data. The concept is simple, make use of the existing technology to expand to different organisms, while retaining all functionalities of the framework. While this serves two purposes, testing the limits of the framework and contributing to the scientific field of synthetic biology, there are some concerns that should be addressed regardless of the framework and organism.

One of the limiting factors in the annotation of organisms is the use of proper identifiers. Whereas in other fields of technology the use of identifiers is mostly used to establish internal relationships and entries, in biological databases it is crucial to find a commonly used identifier that allows scientists to always know which element they are referring to. This is a major part of data integration and contextualization. As discussed before, without proper biological context, the data is indeed meaningless and this is particularly important for this organism, as with its scarcity it is imperative to always

keep track of all information. As this organism derives from *M. mycoides*, it would be fair and rational to assume that JCVI-syn3.A inherits all of its annotation. However, its parent is equally poorly understood and thus, lacks annotation. Hence, the need to have as little liabilities as possible when building a database from scratch for an organism with close to no information is crucial.

Having this in mind, any database will have its foundation for a relational model set. The general rule for this is to preserve any unique identifiers already set, if they exist, or create some that will always be used in the future, not only by the developers, but also by the community. A good example of this, and widely used in different databases, are the locus tags. These are known by community and will always be specific to the gene in question, allowing to even use them as search index. *Syn*Wiki was built around these identifiers, which were already set by the creators of the organism (Hutchison et al., 2016).

Data can be a double-edge sword, both too much and too little can represent a limitation. JCVI-syn3.A can be accounted for both, as obviously its heavily limited data can impose hurdles on the research of scientists but can also be seen as an opportunity. The *Syn*Wiki project aimed to explore this opportunity side, by putting together all available information with the already developed pipelines for the homology analysis and the overall inference from other organisms. With this, it tries to provide the growing community with some more insights on this microorganism. Providing manually curated data will always be a top priority for this Wiki and thus, it is now in the position to offer extensive information on gene and protein levels, access to tailored functional categories and provides a representation for the essentiality of the present genes, while supporting all of the data with relevant literature.

Importantly, *Syn*Wiki was created anticipating the explosion of data in a near future from the field of synthetic biology. Although not populated yet, the platform is prepared to receive and represent information on multiple levels, such as expression (transcriptomics and proteomics), interactions and regulations. As more scientists work with this organism, the objective is to have the platform ready to implement any emerging information, for example the recent expression data in early growth phase (Breuer et al., 2019).

S*yn*Wiki is an open chapter with room and space to grow as fast as the data generation. By taking full advantage of some of the core features of its parent framework, *Subti*Wiki v4, it is growing in popularity among the synthetic biology community and it is expected that this platform will continue expanding in data and attracting more scientist around the globe.

6.5. *Core*Wiki – a modern framework for future Wikis

*Core*Wiki was built on the idea to address most issues encountered in the development of the current *Subti*Wiki framework. Maintainability is a crucial aspect of any software application and biological databases are no different. Changing the backend framework from custom-made to Flask addresses these hurdles and it now provides with the proper documentation to support the development and maintenance of the Wiki databases. The present work focused mostly on the backend framework itself, while the frontend experienced minor improvements.

Flask provides a modular architecture, which helps to ensure that the Wikis excels on the modifiability level and maintainability. The framework is overall less complex than its predecessor resulting in an easier way to implement novel features. To support this, the re-evaluated database plays a major role. It counts with the implementation of an improved relational database **schema**, in which each model allows to set up a clear and intuitive relationships with other models. To complement the new models, there are restrictions when using JSON columns, allowing to access and manipulate information easily as well as removing some of the heavy load from the client side. This is expected to improve overall performance since instead of loading a whole JSON entry with unnecessary high amounts of information, the platform can select which information to load from the database on the backend. Finally, Jinja2 templates allow to organise and display the information sent by the backend in building blocks.

Since most prokaryotes share the same biological entities, i.e., genes are the central elements of the information, *Core*Wiki was built not only aiming to replace the current framework of the Wikis, but also to serve as a starting point for every other microorganism. With some programming knowledge, any developer can take *Core*Wiki and change it to better fit their own needs and organisms. The relationships established between central elements, e.g., genes and proteins, genes and synonyms, genes and categories, are preserved in *Core*Wiki and very well implemented, allowing some flexibility in its contents, which should be imperative when considering expanding to other organisms.

6.6. The *Core*Wiki database

*Core*Wiki excels on the organisation and storage of information while retaining its intrinsic and complex relationships. While the current *Subti*Wiki database relies heavily on JSON entries, in this novel framework it keeps these relationships together and refer to JSON only as a last resource, when no relationship is necessary and maximum flexibility is required. Information on genes and proteins are kept separately, but the models establish relationships automatically. Although the *Subti*Wiki database aims at using the relational model, the heavy use of JSON entries nullifies this purpose. In this **schema**,

the **Gene** table (Figure 6.3A) establishes very few relationships: **ParalogousProteins**, **GeneCategory**, **Sequence**, **Interaction**, **OmicsData_gene** and **MaterialViewGeneRegulation**. Because JSON is heavily used, only physical entries require an actual relationship, causing for most tables to have no relationship when in theory they should, e.g., a gene and its operon. This was addressed in *Core*Wiki and it achieved a much better relational model, efficiently establishing relationships between most of data, where the loose tables are only product of independent management, e.g., **User** table. Moreover, all tables have, directly or indirectly, a relationship with the **Gene** table, reinforcing the relational model and the centrality of the **Gene** entity (Figure 6.3B).

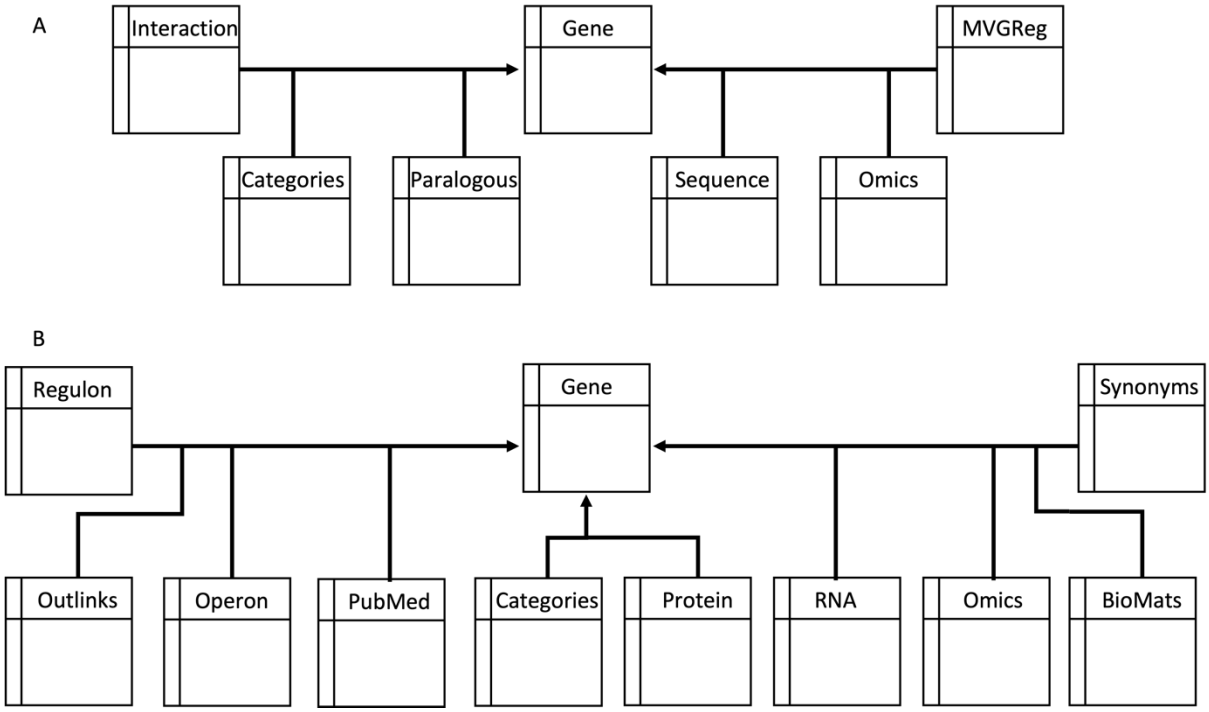


Figure 6.3 – Comparison between *Subti*Wiki v4 (A) and *Core*Wiki (B) **schema** relationships established with the respective **Gene** table**.**

On a more detailed look, the models developed in *Core*Wiki also contain more information that is specific to the entity they represent. For example, in *Subti*Wiki v4 **schema**, information regarding the protein and RNA elements are fully included in the **Gene** table under the JSON entry. Instead, *Core*Wiki separated these two entities, and all information belonging to them was organised. For instance, molecular weight, isoelectric point, protein family, protein structure, localization and protein domains refer only to the **Protein** model, while information on RNA structure is only specific to the **RNA** model. As a consequence of this, it is possible to reach all information pertaining to a single entity, including all relations in different tables through the use of relationships.

Another notable change targets the storage of references. In the previous model, *Subti*Wiki v4 makes use of the **PubMed** table, which was decided to be kept. By establishing a relationship with the **Gene** table, it is possible to now move away from the previous way of storing the data, where each entry was stored as HTML. This is a great example of how the *Core*Wiki database is superior to the one found in *Subti*Wiki v4, where taking the same model but restructuring it to better fit the more recent one can help to access and manipulate the data. The new structure and data organisation ensures that saving information and loading it on client-side are a much lighter task handled by the framework. Another example of better model design are the new Wiki pages. Before, these pages were stored in individual single database entry that would contain all information regarding all pages. Transforming the data to be included in multiple tables, as described here, makes full use of the presented database structure to better handle the information.

6.7. Outlook

*Subti*Wiki is growing stronger than ever, reinforcing its status of prime resource among the *B. subtilis* community. With more emerging technology, it is mandatory to have a prepared framework to host and display novel data. With its current limitations, *Subti*Wiki v4 framework cannot withstand the growing complex information. Not only *Subti*Wiki, but every database developed under this framework, as it is the case for *Syn*Wiki, will have major challenges addressing this issue. Although *Core*Wiki is still under development, it is safe to assume its internal deployment in an intranet server is soon bound to happen for further testing. Migrating the data from our databases should also pose no challenge as Flask-SQLAlchemy takes care of the heavy computing and querying for the developer. One of the major testing and development hallmarks will be the implementation of multiple strains, a feature requested by the scientific community that aims to provide better and more information for *B. subtilis* but can also be used for other organisms.

Much has been achieved for the novel framework *Core*Wiki, but there is still some work to be done. Core parts of the previous platform are yet to be integrated, such as the Genome Browser, Interaction and the Regulation Browsers. Although these implementations are mostly JavaScript libraries, most of the work when transposing these features from one framework to the other is to change how the information is loaded. In the previous Browsers, the data is loaded based on the previous models, which does not work under the current state, thus some adaptations need to be made. This also leaves the open question to future developers whether it would be relevant to implement these features from scratch or adapt the current implementation to the new framework.

Another relevant aspect to be improved is the current gene editor (Figure 6.4). The current version is prone to errors because it only contains a single text box where each major section in the JSON virtual column is marked with an asterisk, *, followed by the title of the section, e.g., *** strain**. All entries with no information must be marked in the respective field by **insert text here**, which when changed from this default value, or moved from its place, will cause the page to be somewhat affected, either by breaking it or hiding some elements. To be further considered are some backend validation processes which will ensure that the information that gets passed end to end is viable and in the correct format, freeing the user from the responsibility of wrongfully change something.

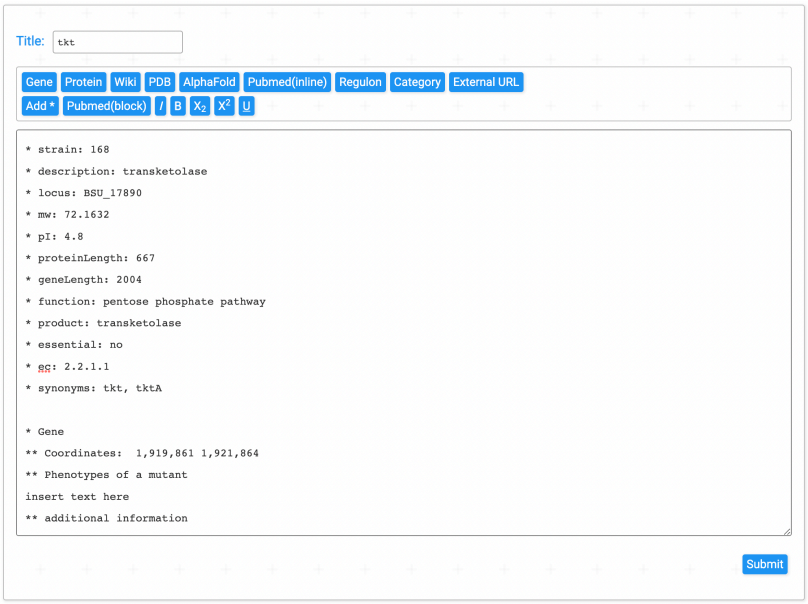


Figure 6.4 – Editor from the current *Subti*Wiki framework using *tkt* gene as example. Each major section is labelled with an asterisk, *, and if no data is available **insert text here** placeholder must be used.

With *Core*Wiki in motion, it is important to think of future implementations and how to accommodate novel data. As mentioned in section 5.4.3.2, the model for metabolite-protein interactions was here implemented, and although the framework is ready for the data and relationships, there is still the need to implement such feature on the frontend (Interaction Browser). An ongoing development approach tries to load it altogether with the current data on interactions and have a button to simulate the binding of the metabolites (Figure 6.5). The idea is to load the metabolites that are documented to interact with a protein of interest on demand, and by choosing to display the interactions, the user will be able to see its effect.

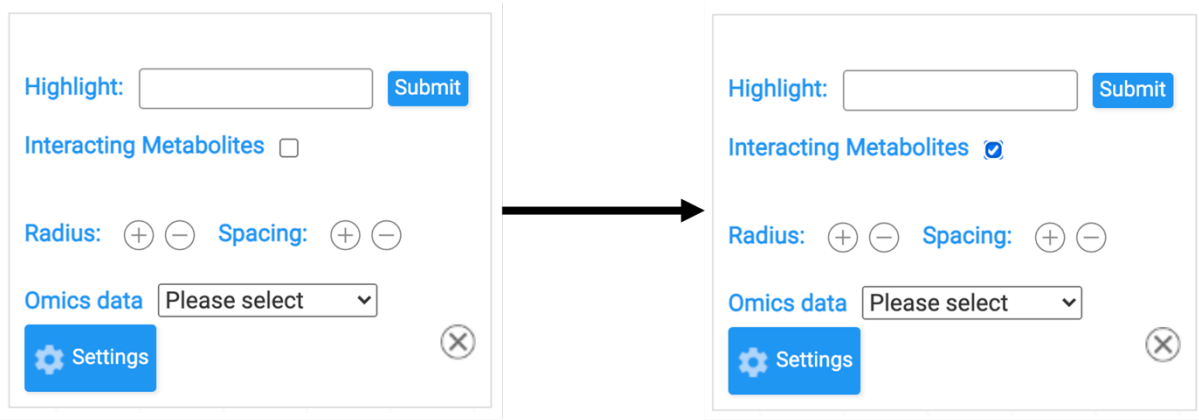


Figure 6.5 – Early suggestion of potential implementation of metabolite-protein interactions in the Interaction Browser. A checkbox allows to simulate the interaction between the loaded metabolite with target protein and, consequently, its effect.

A recent major scientific contribution has allowed to predict how proteins fold (Jumper et al., 2021). AlphaFold allows scientists to predict protein structures of a given amino acid sequence with high accuracy. This major breakthrough is revolutionizing bioinformatics as we know it because now, we can take proteins with unknown structure and look at them with different eyes, stretching further into the functional classification. This is particularly relevant for any organism as, even among model organisms, there is still a big fraction of protein structures to be unveiled. For example, in *Subti*Wiki, around 30% of the genes are still annotated with "Unknown Function", and the same can be seen for around 19% of genes in *Syn*Wiki, which is also a reflection of its poorly understanding in all levels.

Recently, the database for AlphaFold was also released (Varadi et al., 2022), providing every user with an API to extract bulks of information. This new database includes information on several organisms, with the prediction of protein structure for virtually all proteins. Among these, *B.subtilis* and *M.pneumoniae* can be found and thus, enables to fully extract the protein prediction for every protein in *Subti*Wiki and *Myco*Wiki. Since *Core*Wiki aims to replace not only *Subti*Wiki but also every other Wiki created in the lab, this is a major turning point to adding more high-quality content to these platforms. For these organisms it is possible to run the API to extract the necessary files and quickly implement them in these platforms. For the remaining Wikis, it is possible to make full use of the AlphaFold algorithm to manually predict every structure, or simply use recently developed algorithms, such as ColabFold, that are faster but still rely on AlphaFold to predict structures (Mirdita et al., 2022), with the potential for further comparison with other similar structures (van Kempen et al., 2022). Figure 6.6 shows the suggestion for the already ongoing pipeline to address this.

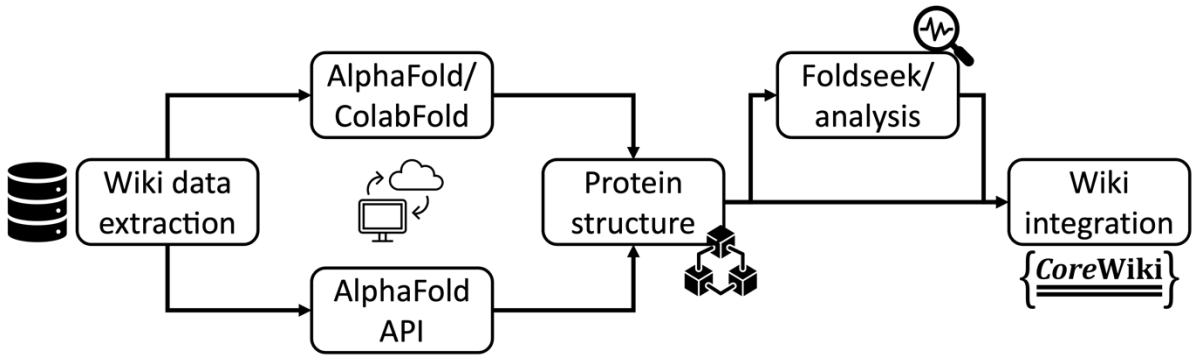


Figure 6.6 – Pipeline for AlphaFold protein prediction integration. Wiki data extraction step can be interpreted as either extracting information of protein sequence, if running AphaFold or ColabFold manually, or extracting UniProt identifiers to run with the AlphaFold API.

Furthermore, having full control over the protein structure also enables to integrate more information on the protein level. Coupling this with experimental data, such as crosslinking, will enable the user to have a novel representation of this information overlayed in the predicted protein structure. This will help further expand and establish the support that *Core*Wiki wants to achieve for every member of the community.

On a different note, it is also possible deploy machine learning algorithms in the backend of *Core*Wiki. Although there is the awareness of the sensitivity of each scientist's research and respect their privacy and secrecy, machine learning in an anonymous and personalized way can be used to boost user engagement, potentially offering suggestions of content based on their preferences. For example, if a specific user accesses a gene page often, but does not make full use of the available features, e.g., Expression Browser, then the platform can politely suggest visiting such section to improve their own research.

Integrating more and better information is a hard accomplishment to aim for. However, the idea behind *Core*Wiki strives to easily address some of the major constraints behind the development of a biological database. Additionally, loading and displaying data in this novel platform becomes almost trivial, giving scientists an easy way of building their own platform as long as they follow certain data format rules. Thus, it is to be predicted that *Core*Wiki will be released in a near future, revolutionizing biological databases, providing with a better and more robust framework that is equipped with the possibility to expand beyond the current organisms. Moreover, with features such as the metabolite-protein interaction, AlphaFold structure integration and multiple strain visualization, it hopes to attract more scientists to visit all inherent Wikis, where everyone can browse curated data on an intuitive and captivating web page.

Chapter 7 – References

Agarwala, R., Barrett, T., Beck, J., Benson, D. A., Bollin, C., *et al*. (2016). Database resources of the National Center for Biotechnology Information. *Nucleic Acids Research*, **44**: D7–D19.

Alkan, C., Sajjadian, S., and Eichler E. E. (2011). Limitations of next-generation genome sequence assembly. *Nature Methods* **8**: 61–65.

Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *J. Mol. Biol*, **215**: 403–410.

Apache with Flask (2016). https://flask.palletsprojects.com/en/2.0.x/deploying/mod_wsgi/ - Retrieved April 28, 2022.

Arnaouteli, S., Bamford, N. C., Stanley-Wall, N. R., and Kovács, Á. T. (2021). *Bacillus subtilis* biofilm formation and social interactions. *Nature Reviews Microbiology* **19**: 600–614.

Ashburner, M., Ball, C. A., Blake, J. A., Botstein, D., Butler, H., *et al*. (2000). Gene ontology: tool for the unification of biology. *Nature Genetics* **25**: 25–29.

Axmark, D., and Widenius, M (2022). *MySQL 5.7 reference manual*. http://dev.mysql.com/doc/refman/5.7/en/index.html - Retrieved April 28, 2022.

Bakken, S. S., Suraski, Z., and Schmid, E. (2000). PHP manual. *iUniverse, Incorporated*, **1**.

Bateman, A., Martin, M. J., Orchard, S., Magrane, M., Agivetova, R., *et al*. (2021). UniProt: The universal protein knowledgebase in 2021. *Nucleic Acids Research*, **49**: D480–D489.

Baxevanis, A. D., and Bateman, A. (2015). The importance of biological databases in biological discovery. *Current Protocols in Bioinformatics*, **50**: 1.1.1–1.1.8.

Bayer, M. (2012). SQLAlchemy. *The architecture of open source applications volume II: structure, scale, and a few more fearless hacks,* **2**.

Benda, M., Woelfel, S., Faßhauer, P., Gunka, K., Klumpp, S., *et al*. (2021). Quasi-essentiality of RNase Y in *Bacillus subtilis* is caused by its critical role in the control of mRNA homeostasis. *Nucleic Acids Research*, **49**: 7088–7102.

Benson, D. A., Cavanaugh, M., Clark, K., Karsch-Mizrachi, I., Lipman, D. J., *et al*. (2013). GenBank. *Nucleic Acids Research*, **41**: D36–D42.

Berardini, T. Z., Reiser, L., Li, D., Mezheritsky, Y., Muller, R., *et al*. (2015). The *Arabidopsis* information resource: making and mining the "gold standard" annotated reference plant genome. *Genesis*, ***53***: 474–485.

Berman, H. M., Westbrook, J., Feng, Z., Gilliland, G., Bhat, T. N., *et al*. (2000). The Protein Data Bank. *Nucleic Acids Research*, **28**: 235–242.

Biggs, N. L., Lloyd, K. E., and Wilson, R. J. (1976). Graph theory 1736-1936. Clarendon Press **1**.

Bond, M., Holthaus, S. M. kleine, Tammen, I., Tear, G., and Russell, C. (2013). Use of model organisms for the study of neuronal ceroid lipofuscinosis. *Biochimica et Biophysica Acta - Molecular Basis of Disease*, **1832**: 1842–1865.

Breuer, M., Earnest, T. M., Merryman, C., Wise, K. S., Sun, L., *et al*. (2019). Essential metabolism for a minimal cell. *ELife*, **8**: 1–75.

Bult, C. J., Blake, J. A., Smith, C. L., Kadin, J. A., Richardson, J. E., *et al*. (2019). Mouse Genome Database (MGD) 2019. *Nucleic Acids Research*, **47**: D801–D806.

Cameron, D. E., Bashor, C. J., and Collins, J. J. (2014). A brief history of synthetic biology. *Nature Reviews Microbiology*, **12**: 381–390.

Caswell, J., Gans, J. D., Generous, N., Hudson, C. M., Merkley, E., *et al*. (2019). Defending our public biological databases as a global critical infrastructure. *Frontiers in Bioengineering and Biotechnology*, ***7***: 58.

Cherry, J. M., Hong, E. L., Amundsen, C., Balakrishnan, R., Binkley, G., *et al*. (2012). *Saccharomyces* Genome Database: The genomics resource of budding yeast. *Nucleic Acids Research*, **40**: D700–D705.

Chinnici, R., Haas, H., Lewis, A. A., Moreau, J.-J., Orchard, D., and Weerawarana, S (2007). *Web services description language (WSDL) version 2.0 part 2: adjuncts.* https://www.w3.org/TR/wsdl20/#Endpoint - Retrieved April 28, 2022.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun ACM*, **13**: 377–387.

Commichau, F. M., Pietack, N., and Stülke, J. (2013). Essential genes in *Bacillus subtilis*: A re-evaluation after ten years. In *Molecular BioSystems*, **9**: 1068–1075.

Corley, M., Burns, M. C., and Yeo, G. W. (2020). How RNA-binding proteins interact with RNA: molecules and mechanisms. *Molecular Cell*, **78**: 9–29.

Dietrich, M. R., Ankeny, R. A., and Chen, P. M. (2014). Publication trends in model organism research. *Genetics*, **198**: 787–794.

Errington, J., and van der Aa, L. T. (2020). Microbe profile: *Bacillus subtilis*: model organism for cellular development, and industrial workhorse. *Microbiology (United Kingdom)*, **166**: 425–427.

Estdale, J., and Georgiadou, E. (2018). Applying the ISO/IEC 25010 quality models to software product. *Communications in Computer and Information Science*, **896**: 492–503.

Fan, S. H., Ebner, P., Reichert, S., Hertlein, T., Zabel, S., *et al*. (2019). MpsAB is important for *Staphylococcus aureus* virulence and growth at atmospheric CO2 levels. *Nature Communications*, **10**: 3627.

Fan, S.-H., Matsuo, M., Huang, L., Tribelli, P. M., and Götz, F. (2021). The MpsAB bicarbonate transporter is superior to carbonic anhydrase in biofilm-forming bacteria with limited CO 2 diffusion. *Microbiology Spectrum*, **9**: e0030521.

Fields, S., and Johnston, M. (2005). Cell biology. Whither model organism research? *Science*, **307**: 1885-1886.

Figaro, S., Durand, S., Gilet, L., Cayet, N., Sachse, M., and Condon, C. (2013). *Bacillus subtilis* mutants with knockouts of the genes encoding ribonucleases RNase Y and RNase J1 are viable, with major defects in cell morphology, sporulation, and competence. *Journal of Bacteriology*, **195**: 2340–2348.

Flanagan, D. (2011). JavaScript: the definitive guide. *O'Reilly*, **6**.

Flórez, L. A., Roppel, S. F., Schmeisky, A. G., Lammers, C. R., and Stülke, J. (2009). A community-curated consensual annotation that is continuously updated: The *Bacillus subtilis* centred wiki *Subti*Wiki. *Database*, **2009**: bap013.

Fruchterman, T. M. J., and Reingold, E. M. (1991). Graph drawing by force-directed placement. *Software - Pract. Exper.*, **21**: 1129–1164.

Fukuda, A., Kodama, Y., Mashima, J., Fujisawa, T., and Ogasawara, O. (2021). DDBJ update: streamlining submission and access of human data. *Nucleic Acids Research*, **49**: D71–D75.

Galperin, M. Y., Wolf, Y. I., Makarova, K. S., Alvarez, R. V., Landsman, D., and Koonin, E. v. (2021). COG database update: focus on microbial diversity, model organisms, and widespread pathogens. *Nucleic Acids Research*, **49**: D274–D281.

Garner, K. L. (2021). Principles of synthetic biology. *Essays in Biochemistry*, **65**: 791–811.

Gerovac, M., Vogel, J., and Smirnov, A. (2021). The world of stable ribonucleoproteins and its mapping with grad-seq and related approaches. *Frontiers in Molecular Biosciences*, **8**:661448.

Grinberg, M. (2018). *Flask web development: developing web applications with python*. O'Reilly Media, Inc, **2**.

Hickson, I., Pieters, S., van Kesteren, A., Jägenstedt, P., and Denicola, D (2022). *HTML living standard*. https://html.spec.whatwg.org/ - Retrieved April 28, 2022.

Hipp, R. D. *SQLite* (2020). https://www.sqlite.org/index.html - Retrieved April 28, 2022.

Hutchison, C. A., Chuang, R. Y., Noskov, V. N., Assad-Garcia, N., Deerinck, T. J., *et al*. (2016). Design and synthesis of a minimal bacterial genome. *Science*, ***351***: aad6253.

Inamine, J. M., Ho, K.-C., Loechel, S., and Hul, P.-C. (1990). Evidence that UGA is read as a tryptophan codon rather than as a stop codon by *Mycoplasma pneumoniae*, *Mycoplasma genitalium*, and *Mycoplasma gallisepticum*. *Journal of Bacteriology*, **172**: 504–506.

Ishii, T., Yoshida, K.-I., Terai, G., Fujita, Y., and Nakai, K. (2001). DBTBS: a database of *Bacillus subtilis* promoters and transcription factors. *Nucleic Acids Research,* **29**: 278–280.

JSON Formatter and Validator (2007). https://jsonformatter.curiousconcept.com/# - Retrieved April 28, 2022.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., *et al*. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, ***596***: 583–589.

Kanehisa, M., Furumichi, M., Sato, Y., Ishiguro-Watanabe, M., and Tanabe, M. (2021). KEGG: integrating viruses and cellular organisms. *Nucleic Acids Research*, **49**: D545–D551.

Kanz, C., Aldebert, P., Althorpe, N., Baker, W., Baldwin, A., *et al*. (2005). The EMBL nucleotide sequence database. *Nucleic Acids Research*, **33**: D29–D33.

Karp, P. D., Billington, R., Caspi, R., Fulcher, C. A., Latendresse, M., *et al*. (2018). The BioCyc collection of microbial genomes and metabolic pathways. *Briefings in Bioinformatics*, ***20***: 1085–1093.

Keseler, I. M., Gama-Castro, S., Mackie, A., Billington, R., Bonavides-Martínez, C., *et al*. (2021). The EcoCyc database in 2021. *Frontiers in Microbiology*, **12**: 711077.

Khalil, A. S., and Collins, J. J. (2010). Synthetic biology: Applications come of age. *Nature Reviews Genetics*, **11**: 367–379.

Klewing, A., Koo, B. M., Krüger, L., Poehlein, A., Reuß, D., *et al*. (2020). Resistance to serine in *Bacillus subtilis*: identification of the serine transporter YbeC and of a metabolic network that links serine and threonine metabolism. *Environmental Microbiology*, **22**: 3937–3949.

Koo, B. M., Kritikos, G., Farelli, J. D., Todor, H., Tong, K., *et al*. (2017). Construction and analysis of two genome-scale deletion libraries for *Bacillus subtilis*. *Cell Systems*, ***4***: 291–305.

Kovács, Á. T. (2019). *Bacillus subtilis*. *Trends in Microbiology*, **27**: 724–725.

Krasner, G. E., and Pope, S. T. (1988). A cookbook for using view-controller user the model-interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 26–49.

Krüger, L., Herzberg, C., Rath, H., Pedreira, T., Ischebeck, T., *et al*. (2021). Essentiality of c-di-AMP in *Bacillus subtilis*: bypassing mutations converge in potassium and glutamate homeostasis. *PLoS Genetics*, ***17***: e1009092.

Lachance, J., Matteau, D., Brodeur, J., Lloyd, C. J., Mih, N., *et al*. (2021). Genome-scale metabolic modeling reveals key features of a minimal gene set. *Molecular Systems Biology*, **17**: e10099.

Larkin, A., Marygold, S. J., Antonazzo, G., Attrill, H., dos Santos, G., *et al*. (2021). FlyBase: updates to the *Drosophila melanogaster* knowledge base. *Nucleic Acids Research*, **49**: D899–D907.

Letunic, I., and Bork, P. (2021). Interactive tree of life (iTOL) v5: An online tool for phylogenetic tree display and annotation. *Nucleic Acids Research*, **49**: W293–W296.

Lie, H. W., and Bos, B. (2005). Cascading Style Sheet – designing for the Web. *Addison-Wesley*, **3**.

Link, H., Kochanowski, K., and Sauer, U. (2013). Systematic identification of allosteric protein-metabolite interactions that control enzyme activity in vivo. *Nature Biotechnology*, **31**: 357–361.

Loman, N. J., Misra, R. v., Dallman, T. J., Constantinidou, C., Gharbia, S. E., *et al* (2012). Performance comparison of benchtop high-throughput sequencing platforms. *Nature Biotechnology*, **30**: 434–439.

Lorenz, M., Hesse, G., Rudloph, J.-P., Uflacker, M., De, M. U., and Plattner, H. (2017). Object-relational mapping reconsidered a quantitative study on the impact of database technology on O/R mapping strategies. *Proceedings of the 50th Hawaii International Conference on System Sciences*, 4877–4886.

Madeira, F., Park, Y. M., Lee, J., Buso, N., Gur, T., *et al*. (2019). The EMBL-EBI search and sequence analysis tools APIs in 2019. *Nucleic Acids Research*, **47**: W636–W641.

Mäder, U., Schmeisky, A. G., Flórez, L. A., and Stülke, J. (2012). *Subti*Wiki - A comprehensive community resource for the model organism *Bacillus subtilis*. *Nucleic Acids Research*, ***40***: D1278–1287.

Manzoni, C., Kia, D. A., Vandrovcova, J., Hardy, J., Wood, N. W., Lewis, P. A., and Ferrari, R. (2018). Genome, transcriptome and proteome: the rise of omics data and their integration in biomedical sciences. *Briefings in Bioinformatics*, **19**: 286–302.

Mardis, E. R. (2017). DNA sequencing technologies: 2006-2016. *Nature Protocols*, **12**: 213–218.

McCool, R (1999). *Apache HTTP server project*. https://httpd.apache.org/docs/2.4/en/mod/ - Retrieved April 28, 2022.

MediaWiki (2022). https://www.mediawiki.org/wiki/MediaWiki - Retrieved April 28, 2022.

Michalik, S., Reder, A., Richts, B., Faßhauer, P., Mäder, U., *et al*. (2021). The *Bacillus subtilis* minimal genome compendium. *ACS Synthetic Biology*, **10**: 2767–2771.

Michna, R. H., Commichau, F. M., Tödter, D., Zschiedrich, C. P., and Stülke, J. (2014). *Subti*Wiki-A database for the model organism *Bacillus subtilis* that links pathway, interaction and expression information. *Nucleic Acids Research*, **42**: D692–698.

Michna, R. H., Zhu, B., Mäder, U., and Stülke, J. (2016). *Subti*Wiki 2.0 - An integrated database for the model organism *Bacillus subtilis*. *Nucleic Acids Research*, **44**: D654–D662.

Microsoft (2021). *SQL injection*. https://docs.microsoft.com/en-us/sql/relational-databases/security/sql-injection?view=sql-server-ver15 - Retrieved April 28, 2022.

Mirdita, M., Schütze, K., Moriwaki, Y., Heo, L., Ovchinnikov, S., and Steinegger, M. (2022). ColabFold-Making protein folding accessible to all. *BioRxiv*, *08.15.456425*.

Mönnich, A., Ronacher, A., Lord, D., Li, G., Bronson, J., *et al* (2016). *The Pallets Project*. https://palletsprojects.com/ - Retrieved April 28, 2022.

Monteiro, P. T., Oliveira, J., Pais, P., Antunes, M., Palma, M., *et al*. C. (2020). YEASTRACT+: A portal for cross-species comparative genomics of transcription regulation in yeasts. *Nucleic Acids Research*, **48**: D642–D649.

Moszer, I., Glaser, P., & Danchin, A. (1995). SubtiList: a relational database for the *Bacillus subtilis* genome. *Microbiology*, **141**: 261–268.

Moszer, I., Jones, L. M., Moreira, S., Fabry, C., and Danchin, A. (2002). SubtiList: the reference database for the Bacillus subtilis genome. *Nucleic Acids Research*, **30**: 62-65.

National Center For Biotechnology Information. (2010). *Entrez programming utilities help*. https://www.ncbi.nlm.nih.gov/books/NBK25501/ - Retrieved April 28, 2022.

Niehaus, T. D., Elbadawi-Sidhu, M., de Crécy-Lagard, V., Fiehn, O., and Hanson, A. D. (2017). Discovery of a widespread prokaryotic 5-oxoprolinase that was hiding in plain sight. *Journal of Biological Chemistry*, **292**: 16360–16367.

O'Connor, B. D., Day, A., Cain, S., Arnaiz, O., Sperling, L., and Stein, L. D. (2008). GMODWeb: A web framework for the generic model organism database. *Genome Biology*, **9**: R102.

Ogiwarea, A., Ogasawara, N., Watanabe, M., and Takagi, T. (1996). Construction of the *Bacillus subtilis* ORF database (BSORF DB). *Genom. Inform.*, **7**: 228–229.

Oliver, S. G., Lock, A., Harris, M. A., Nurse, P., and Wood, V. (2016). Model organism databases: essential resources that need the support of both funders and users. *BMC Biology*, **14**: 14-49.

O'Reilly, F. J., Xue, L., Graziadei, A., Sinn, L., Lenz, S., *et al*. (2020). In-cell architecture of an actively transcribing-translating expressome. *Science*, **369**: 554–557.

Oughtred, R., Rust, J., Chang, C., Breitkreutz, B. J., Stark, C., *et al*. (2021). The BioGRID database: A comprehensive biomedical resource of curated protein, genetic, and chemical interactions. *Protein Science*, **30**: 187–200.

Patel, K. (2013). Incremental journey for World Wide Web: introduced with Web 1.0 to recent Web 5.0—a survey paper. *International Journal of Advanced Research in Computer Science and Software Engineering*, **3**: 410–417.

Pedreira, T., Elfmann C., and Stülke J. (2022). The current state of *Subti*Wiki, the database for the model organism *Bacillus subtilis*. *Nucleic Acids Research*, **50**: D875–D882.

Reuß, D. R., Altenbuchner, J., Mäder, U., Rath, H., Ischebeck, T., *et al*. (2017). Large-scale reduction of the *Bacillus subtilis* genome: consequences for the transcriptional network, resource allocation, and metabolism. *Genome Research*, **27**: 289–299.

Reuß, D. R., Commichau, F. M., Gundlach, J., Zhu, B., and Stülke, J. (2016). The blueprint of a minimal cell: MiniBacillus. *Microbiology and Molecular Biology Reviews*, **80**: 955–987.

Reuß, D. R., Faßhauer, P., Mroch, P. J., Ul-Haq, I., Koo, B. M., *et al*. (2019). Topoisomerase IV can functionally replace all type 1A topoisomerases in *Bacillus subtilis*. *Nucleic Acids Research*, **47**: 5231–5242.

Reuter, J. A., Spacek, D. v., and Snyder, M. P. (2015). High-throughput sequencing technologies. *Molecular Cell*, **58**: 586–597.

Rocha, E. P. C., Danchin, A., and Viari, A. (1999). Translation in *Bacillus subtilis*: roles and trends of initiation and termination, insights from a genome analysis. *Nucleic Acids Research*, **27**: 3567–3576.

Sachla, A. J., and Helmann, J. D. (2019). A bacterial checkpoint protein for ribosome assembly moonlights as an essential metabolite-proofreading enzyme. *Nature Communications*, **10**: 1526.

Sass - CSS with superpowers (2022). https://sass-lang.com/ - Retrieved April 28, 2022.

Schwille, P. (2011). Bottom-up synthetic biology: engineering in a tinkerer's World. *Science*, **333**: 1252.

Senges, C. H. R., Stepanek, J. J., Wenzel, M., Raatschen, N., Ay, Ü., *et al*. (2021). Comparison of proteomic responses as global approach to antibiotic mechanism of action elucidation. *Antimicrobial Agents and Chemotherapy*, **65**: e01373–20.

Sidiq, K. R., Chow, M. W., Zhao, Z., and Daniel, R. A. (2021). Alanine metabolism in *Bacillus subtilis*. *Mol. Microbiol.*, **115**: 739–757.

Sierro, N., Makita, Y., de hoon, M., and Nakai, K. (2008). DBTBS: A database of transcriptional regulation in *Bacillus subtilis* containing upstream intergenic conservation information. *Nucleic Acids Research*, **36**: D93–D96.

Stackshare (2022). *Flask using applications*. https://stackshare.io/flask - Retrieved April 28, 2022.

Suárez, R. A., Stülke, J., and van Dijl, J. M. (2019). Less is more: toward a genome-reduced *Bacillus* cell factory for "difficult proteins." *ACS Synthetic Biology*, **8**: 99–108.

Szklarczyk, D., Gable, A. L., Lyon, D., Junge, A., Wyder, S., *et al*. (2019). STRING v11: protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic Acids Research*, **47**: D607–D613.

TIOBE (2022). *TIOBE programming languages index*. https://www.tiobe.com/tiobe-index/ - Retrieved April 28, 2022.

Tödter, D., Gunka, K., and Stulke, J. (2017). The highly conserved Asp23 family protein YqhY plays a role in lipid biosynthesis in *Bacillus subtilis*. *Frontiers in Microbiology*, **8**: 883.

van Kempen, M., Kim, S. S., Tumescheit, C., Mirdita, M., Söding, J., and Steinegger, M. (2022). Foldseek: fast and accurate protein structure search. *BioRxiv*, *02.07.479398*.

van Rossum, G., and Fred L., D. (2002). Python 3 reference manual. *CreateSpace*.

van Tilburg, A. Y., van Heel, A. J., Stülke, J., de Kok, N. A. W., Rueff, A. S., and Kuipers, O. P. (2020). Mini *Bacillus* PG10 as a convenient and rffective production host for lantibiotics. *ACS Synthetic Biology*, **9**: 1833–1842.

Varadi, M., Anyango, S., Deshpande, M., Nair, S., Natassia, C., *et al*. (2022). AlphaFold protein structure database: massively expanding the structural coverage of protein-sequence space with high-accuracy models. *Nucleic Acids Research*, **50**: D439–D444.

Wicke, D., Schulz, L. M., Lentes, S., Scholz, P., Poehlein, *et al*. (2019). Identification of the first glyphosate transporter by genomic adaptation. *Environmental Microbiology*, **21**: 1287–1305.

Yamao, F., Muto, A., Kawauchi, Y., Azumi, Y., and Osawa, S. (1985). UGA is read as tryptophan in *Mycoplasma capricolum*. *Proc. Nati. Acad. Sci. USA*, **82**: 2306–2309.

Zhang, C., Zheng, W., Cheng, M., Omenn, G. S., Freddolino, P. L., and Zhang, Y. (2021). Functions of essential genes and a scale-free protein interaction network revealed by structure-based function and interaction prediction for a minimal genome. *Journal of Proteome Research*, **20**: 1178–1189.

Zhu, B., and Stülke, J. (2018). *Subti*Wiki in 2018: From genes and proteins to functional network annotation of the model organism *Bacillus subtilis*. *Nucleic Acids Research*, **46**: D743–D748.

Zweers, J. C., Barák, I., Becher, D., Driessen, A. J. M., Hecker, M., *et al*. (2008). Towards the development of *Bacillus subtilis* as a cell factory for membrane proteins and protein complexes. *Microbial Cell Factories*, **7**: 10.

Chapter 8 – Supplementary materials

8.1. Plasmids table

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
name	Varchar	Name of plasmid; Index
description	Text	JSON enabled
pubmed	Text	Publications annotated to plasmid
additional_information	Text	JSON enabled

8.2. Protein complexes table

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
name	Varchar	Name of complex; Index
description	Text	JSON enabled
pubmed	Text	Publications annotated to complex
additional_information	Text	JSON enabled

8.3. Protein domain table

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
name	Varchar	Name of domain; Index
description	Text	JSON enabled
pubmed	Text	Publications annotated to domain
additional_information	Text	JSON enabled

8.4. Protein families table

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
name	Varchar	Name of family; Index
description	Text	JSON enabled
family_members	Text	List of members of protein complex
pubmed	Text	Publications annotated to protein family
type	Text	Annotated type of protein family
additional_information	Text	JSON enabled

8.5. Transcription factors table

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
name	Varchar	Name of transcription factor family; Index
Family_members	Text	List of members of transcription factor family
description	Text	JSON enabled
pubmed	Text	Publications annotated to transcription factor
additional_information	Text	JSON enabled

8.6. Interaction table

Column Name	Data structure	Overview
id	Integer	Primary Key and Unique
description	Text	JSON enabled

8.7. Protein2Interaction table

Column Name	Data structure	Overview
protein_id	Integer	Protein id foreign key
interaction_id	Varchar	Interaction id foreign key

8.8. Protein table

Column Name	Data structure	Overview
id	Integer	Primary key and Unique
gene_id	Varchar	Gene id foreign key
molecular_weight	Numeric	Float value of protein's molecular weight
isoelectric_point	Numeric	Float value of protein's isoelectric point
modification	Text	Known protein modifications
family_id	Text	Protein family id foreign key
structure	Text	Protein structure identifiers
localization	Text	Localization of protein
reaction	Text	Reactions protein is part of
domain_id	Text	Protein domain id foreign key
complex_id	Text	Protein complex id foreign key
additional_information	Text	JSON enabled

8.9. Metabolite2Interaction table

Column Name	Data structure	Overview
metabolite_id	Integer	Metabolite id foreign key
interaction_id	Integer	Interaction id foreign key

Chapter 9 – Curriculum vitae

Tiago Godinho de Ornelas Pedreira

Goßlerstr. 77

37075 Göttingen

Date of Birth: 16.02.1990

Place of Birth: Lisbon, Portugal

Citizenship: Portuguese

PhD study in IMPRS-Genome Science programme

July 2019 – June 2022	Thesis: From annotation to bacterial data models

Other education

2013–2015	Master of Science, Microbiology. University of Lisbon.
2008–2013	Bachelor of Science, Molecular and Cellular Biology. NOVA University

Skills

Languages	Portuguese – Native level
	English – C2 level
	German – A2 level
Computer skills	Programming skills in Python, R, PHP and JavaScript.
	Experience in backend and frontend web development.
	Experience in Bioinformatics analytical pipelines.