University of Minho
School of Engineering
Department of Biological Engineering

Sofia Raquel Lages Pereira

Implementation of the object-relational mapping tool Hibernate in *merlin*'s framework

October 2019

University of Minho
School of Engineering
Department of Biological Engineering

Sofia Raquel Lages Pereira

Implementation of the object-relational mapping tool Hibernate in *merlin*'s framework

Master dissertation
Master Degree in Bioinformatics

Dissertation supervised by
Doctor Oscar Manuel Lima Dias

October 2019

ACKNOWLEDGEMENTS

My biggest thank goes to Paulo, who during these two years of Masters has always supported me, never letting me give up in the most difficult moments. For all the support, sharing of knowledge and patience, I am eternally grateful.

To my family that support me in this decision and gave me strength to fight for my goals.

I thank Professor Doctor Oscar Dias, supervisor of this dissertation, for all the support and for having been certified that I would end this dissertation with much more knowledge. I also thank to my mentor Ruben Rodrigues by the manifested availability to guide me, as well as to all those who, on an institutional level, somehow accompanied or encouraged me.

To all that, directly or indirectly, made possible the accomplishment of this dissertation, my sincere thanks.

To all, thank you.

Sofia Pereira

STATEMENT OF INTEGRITY

RESUMO

Nas últimas décadas, a biotecnologia evoluiu em torno da aplicação de métodos computacionais. Como resultado, ferramentas computacionais, tal como o *merlin*, surgiram com a capacidade de prever algumas funções celulares de um organismo específico de acordo com modificações no genótipo.

O *merlin* é uma ferramenta que permite a reconstrução de modelos metabólicos à escala genómica para organismos, desde que já tenham seu genoma sequenciado. Contudo, apresenta limitações aos níveis da arquitetura e da gestão da base de dados.

Assim, para melhorar a estrutura e gestão da base de dados do merlin, é necessário implementar uma ferramenta de mapeamento de objetos relacional, como o Hibernate.

Pretendeu-se resolver as limitações do *merlin* através da implementação de interfaces de objetos de acesso à base de dados, que utilizam a ferramenta Hibernate para gerir todas as consultas à base de dados.

Ao longo desta tese, foram criados serviços, que usam os objectos de acesso à base de dados criados.

Nesta tese pretendeu-se implementar o Hibernate na aplicação Java^{TM}*merlin*, aplicando e explorando todas as suas vantagens. Esta implementação teve como objetivo remover as dependências das base de dados MySQL^{TM}e H2, do código central do *merlin*.

Todo o código desenvolvido está disponível no repositório merlin 4.0 e pode ser acedido através do link https://gitlab.bio.di.uminho.pt/merlin4/merlin-hibernate/tree/dev_spereira.

Palavras-chave: criteria API, hibernate, merlin, metabolic models reconstruction, ORM.

ABSTRACT

Over the last decades, biotechnology has evolved within application of computational methods. As result, computational tools, such as *merlin*, emerged with the ability to predict some cellular functions of a specific organism within modifications in the genotype.

merlin is a tool that enables the reconstruction of genome-scale metabolic models for organisms, provided that they have their genome sequenced. But it presents limitations at architecture and database management levels.

So, to improve the structure and management of *merlin*'s database it is necessary to implement a relational object mapping tool, such as Hibernate.

It was pretended to solve *merlin*'s limitations within the implementation of database access object interfaces, that are intended to use Hibernate tool to manage all database queries.

Throughout this thesis, services that call created database access objects were created.

In this thesis it was intended to implement Hibernate in the Java^{TM}application *merlin*, applying and exploiting all its advantages. This implementation aimed to remove the dependencies of the MySQL^{TM}and H2 databases from *merlin* core.

All developed code is available in the *merlin* 4.0 repository and can be accessed through the link https://gitlab.bio.di.uminho.pt/merlin4/merlin-hibernate/tree/dev_spereira.

Keywords: criteria API, hibernate, merlin, metabolic models reconstruction, ORM.

CONTENTS

LIST OF FIGURES

ACRONYMS

A	
AIBENCH	Artificial Intelligent workBench.
API	Application Programming Interface.
B	
BLAST	Basic Local Alignment Search Tool.
BRENDA	Braunschweig Enzyme Database.
D	
DAO	Data Access Object.
DBMS	Database Management System.
E	
EC	Enzyme Commission.
EJB	Enterprise JavaBeans.
G	
GPR	Gene-Protein-Reaction.
GSM	Genome-Scale Metabolic.
GUI	Graphical User Interface.
H	
HMMER	Hidden Markov Models.
HQL	Hibernate Query Language.
I	
ID	identifier.

IDE	Integrated Development Environment.
INCHI KEY	International Chemical Identifier.
J	
JAR	Java ARchive.
JDBC	Java Database Connectivity.
JEE	Java Platform Enterprise Edition.
JNDI	Java Naming and Directory Interface.
JPA	Java Persistence API.
JSE	Java Platform Standard Edition.
JTA	Java Transaction API.
K	
KEGG	Kyoto Encyclopedia of Genes and Genomes.
M	
MERLIN	The Metabolic Model Reconstruction Tool Using Genome-Scale Information.
O	
OID	Object Unique Identifier.
OODBMS	Object-Oriented Database Management System.
ORM	Object-Relational Mapping.
P	
PH	Potential of Hydrogen.
POJO	Plain Old Java Objects.
Q	
QBE	Query By Example.
R	

RAM Random-Access Memory.

S

SBML Systems Biology Markup Language.

SMILES Simplified Molecular Input Line Entry System.

SQL Structured Query Language.

U

UNIPROT The Universal Protein Resource.

X

XML eXtensible Markup Language.

1 INTRODUCTION

This dissertation describes the Master's work developed in the context of the Masters in Bioinformatics held at the Informatics Department and at the Department of Biological Engineering, University of Minho.

1.1 CONTEXT AND MOTIVATION

Systems biology is an interdisciplinary biology-based field of study that focuses on complex interactions within biological systems. It consists in a series of protocols used to do research, containing theory and analytical or computational modeling to test specific hypotheses about a biological system (1).

Systems biology have been applied in different fields such as biological engineering and even medicine. For instance, Evandro and coworkers propose the usage of systems biology to better understand complex cardiovascular diseases (2).

So, it is understood that systems biology has been an area of considerable interest in the last years, proving to be one of the most important tools for health professionals, supported by discoveries and advances, especially in the biomedical field, and bioinformatics is able to provide the computational tools necessary for major discoveries, as example, to allow the reconstruction of *Genome-Scale Metabolic (GSM)* models (3).

GSM models are stoichiometric models of cellular metabolism, intended to incorporate the representation of all metabolic transformations present in an organism. These models predict, *in silico*, the phenotype of microorganisms in different genetic and environmental conditions. These predictions can be used for various metabolic engineering applications, since they allow to focus *in vivo* experiments on methodologies that will, in theory, present better results, reducing the high costs on time and money spent in laboratorial experiments. (4)(5).

The reconstruction of genome-scale metabolic models requires structured knowledge retrieved from biochemical, genetic and genomic fields, containing detailed information about a specific organism. This knowledge forms a basis for the formulation of genomic-scale genotype-phenotype relationships (6).

Given the relevance of these models, computational tools, such as *The Metabolic Model Reconstruction Tool Using Genome-Scale Information (merlin)* (7), with the ability to predict *in silico* the organism's phenotypical behavior, have emerged.

merlin is an open-source Java^{TM}tool that, by providing a variety of tools, allows to reconstruct GSM models for all organisms with a sequenced genome (8). However, it presents limitations at the database architecture and management levels, for instance, this tool is not able to use different database engines (e.g. Posgree or SQLite) other than the ones integrated in the framework. Besides that, merlin does not have a query manager for its interaction with the database. Instead, merlin's interaction with database layer is done using prepared statements with structured query language queries embedded directly into the source code (hard-coded).

Improving the structure and management of merlin's database requires implementing an object-relational mapping framework, such as Hibernate (9).

Hibernate is a Java^{TM}object-relational mapping framework, that facilitates the storage and retrieval of Java^{TM}objects. It allows to represent the database as objects and then use these objects without using structured query language. Hibernate uses Java^{TM}database connectivity *Application Programming Interface (API)* internally to interact with the database. Also, this framework can be configured to connect to any type of relational database and to achieve it, Java^{TM}database connectivity drivers can be used. Then, different database engines can be used.

The implementation of database access object interfaces, creating software model objects that are independent from its database structure, will allow to overcome merlin's limitations. These interfaces will use the Hibernate framework to manage database queries. Thus, hard-coded structured query language queries are not required.

Therefore, through all this methodology it is intended to remove the management of dependencies related with databases from the merlin's core and be used as a novel plug-in for different merlin's storage solutions.

1.2 MAIN AIMS/OBJECTIVES

Given the context specified above, the main target of this thesis is the implementation of the object-relational mapping tool Hibernate in merlin's framework as a novel plug-in for different storage engines. This will allow reaching another goal, which is to remove the management of MySql^{TM}and H2 databases dependencies from the merlin's main code.

In detail, it is intended to address the following scientific/technological objectives:

- Review and study the relevant literature;
- Create a novel plug-in that integrates the data access objects for the currently supported database available in merlin;

- Implement Hibernate services that use the previously created data access objects;
- Enable the novel plug-in to use different storage engines, like MySQL^{TM}, MariaDB, Oracle, H2, PostgreSQL.

1.3 DOCUMENT ORGANIZATION

The document is organized as follows:

- **Chapter 1**: Introduction

  This chapter presents a brief contextualization of the theme and objectives of this dissertation.

- **Chapter 2**: State of the art

  In this chapter a bibliographic review is made on the following topics:

  - Systems biology and bioinformatics approach
  - Genome-scale metabolic models reconstruction
  - merlin
  - Data persistence
  - Data access object
  - Prepared statements
  - Object-relational mapping
  - Object-relational mapping tool Hibernate
  - Criteria API
  - AIBench

- **Chapter 3**: Proposal solution

  This chapter explains the initial analysis process performed to understand the whole structure of the application and how it was developed, as well as a theoretical solution that allows to achieve the main objective of this thesis.

- **Chapter 4**: Practical Approach

  This chapter covers the practical process of the project, from the implementation process to its integration.

  Firstly, the resources required for this project development are addressed, following the stages of the development process, in order of execution. Each stage of the development process is described in detail, and when needed for better understanding, is graphically exemplified.

  In the end, the final application structure post-implementation is presented.

- **Chapter 5**: Conclusion and future work

  Summary of the entire project and final conclusions. Some considerations are also made regarding possible future work.

2

STATE OF THE ART

2.1 SYSTEMS BIOLOGY AND BIOINFORMATICS APPROACH

Over the last decades, have been trying to understand the relationships in networks of biological processes. The study of the interactions between components of biological systems and how these interactions give rise to the function and behavior of a system is a new paradigm called Systems Biology (10).

The world of systems biology is quite complex. Biochemical interactions are complex and all cells have thousands of interactions, as result of evolution, which through random mutations, selects organisms that survive (11).

According to Kitano and coworkers, it is of the utmost importance to understand the structure and cellular dynamics of the organism, as well as its function. Thus, it is possible to understand biology at the system level (12).

Despite all the advances, due to the intrinsic complexity of biological systems, intuition about their behavior is not enough. Thus, experimental and computational approaches are expected to solve this problem, as these technological advances provide detailed information on networks of biological interactions (13)(11).

In a short time, computer science has become fundamental for the studies of the biological area, opening new frontiers for genomics and proteomics analyzes (3).

Systems biology can be defined as the study of the application of mathematical and computational techniques for the generation and management of biological information (3). This can be divided into two distinct areas: the discovery of knowledge (data mining) and simulation-based analysis. Whereas the first finds patterns in large amounts of experimental data, the second is based on simulation and allows to test hypotheses *in silico*, providing predictions that will be tested *in vitro* (13).

Thus, in order to build and manipulate models, bioinformatics tools can be used. These contribute to advances in drugs discovery, metabolic engineering and even in medicine, since it lead a better understanding of the molecular mechanism of an organism (14)(15).

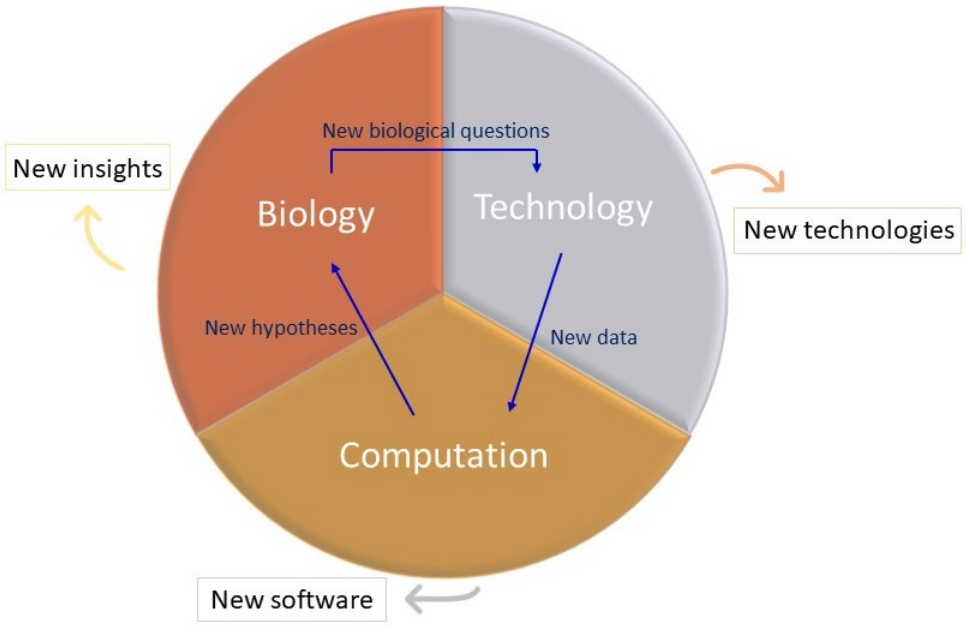


Figure 1.: Biology, technology and computation cycle.

As can be seen from the figure 1, in order to solve the biological problems and explore the huge amount of data, it is necessary to develop new technologies and computacional tools.

Summing up, it can be concluded that the main goal of systems biology is to achieve a deeper understanding of living organisms, and bioinformatic help to achieve this knowledge, since it can provides computacional tools that allow to reconstruct GSM models, simulating the systems network of biochemical reactions of an organism (3).

2.2 GENOME-SCALE METABOLIC MODELS RECONSTRUCTION

In the last decades, the reconstruction and application of GSM models has greatly influenced the area of systems biology, since it allows to characterize and predict the behavior of a microorganism under different environmental and genetic conditions (16)(4).

There are several approaches for reconstructing models, but whatever the approach used for the GSM models reconstruction, it is necessary to know the compounds that works as substrates and products, the stoichiometry, reversibility status and also cellular location, for each of the reactions that belong to the network (17).

Based on a combination of biochemical information and the genomic sequence, this reconstruction provides a basis about which a computational analysis can be performed. It attempts to collect all reactions relative to a target organism, through genome annotation and biochemical knowledge, in order to reconstruct a stoichiometric mathematical model (18).

According to Dias and coworkers, it is possible to associate the organism's genome with its physiology. This is achieved through the identification of biochemical reactions and molecular mechanisms that occur in the organism (4).

Moreover, according Rocha and coworkers, these models can also be used for robustness analysis of a network, by measuring the change in the maximal flux of the objective function when the optimal flux through any particular metabolic reaction is altered (19).

GSM models have been widely used in metabolic engineering studies in order to overcome the knowledge limitations of the existing metabolic network and to identify new metabolic reactions (16).

The GSM model's reconstruction comprises some steps, listed below (20)(18)(21):

1. **Genome annotation.**

   The genome annotation is of most importance as it represents the backbone of the model. In this step, only genes involved in enzyme encoding and transport systems are relevant, so, during the reconstruction, genes that perform such tasks are labeled as metabolic genes.

   To validate the assigned gene functions, *Braunschweig Enzyme Database (BRENDA)* should be used. Then, through the use of tools like *Basic Local Alignment Search Tool (BLAST)* and *Hidden Markov Models (HMMER)*, genomic functional annotation can be obtained.

   Thus, by performing sequence alignment, it may be possible to find homologous genes from other organisms. Based on the similarity score between orthologous genes, it may give an idea of the gene's functionality.

2. **Draft a network reconstruction/assembling metabolic network** from genome-annotation data and information available in the literature (usually from databases such as *Kyoto Encyclopedia of Genes and Genomes (KEGG)* (22) and *BioCyc* (23)).

   After performing the genome annotation and selecting the major metabolic genes, this information will be used to assemble the reactions' network. Others sub-steps like *Gene-Protein-Reaction (GPR)* associations (connections between genes), stoichiometry of the reactions (making sure that everything is chemically balanced), compartmentalization (identifying where the enzyme, and consequently the reaction it promotes, is located within the organism) and manual curation are performed.

3. **Network refinement.**

   Due to errors that may still occur in the network, due for example to annotation errors, it is imperative to correct these errors through manual network investigation. It should be done using, for example, the biochemical literature, comparative genomic approaches and transcriptomic data.

4. **Network conversion into a mathematical/computational model**, according to stoichiometric coefficients.

   After the GSM model is curated, is possible to convert the metabolic network into a stoichiometric model. Then, the biomass formation equation is included as well as other constraints, as example, the reversibility of the reactions.

5. **Model's evaluation and debugging**, through experimentation.

   The model must be experimentally validated and if the model is not in agreement with the data obtained experimentally, it must be carefully reevaluated from the functional annotation.

The main steps of GSM models reconstruction are outlined in figure 2.

Due to discrepancies between *in silico* simulations and wet experimental results, the evaluation and validation can help to improve the accuracy of GSM models. In addition, the last three steps must be repeated iteratively to obtain a higher quality GSM model (18).

Given that the ultimate goal of the reconstruction of a GSM model is to predict reliable engineering targets, these predictions need to be validated in wet laboratory and usually this whole process is comprised of three steps: genetic manipulation, strain cultivation and phenotype measurement (18).

However, due to the large number of compounds involved in many different reactions and pathways, the treatment of all data is not easily manually performed. Recently, GSM models have become more complex, allowing to expand its applicability. Algorithms have been developed and improved to analyze metabolic models and to calculate redistribution of metabolic flux, according to genetic/environmental alterations (16).

Currently, several tools have been developed for reconstructing GSM models, such as Pantograph (24), RAVEN Toolbox (5) and merlin (8). So, the most of the steps in the reconstruction process can be automated with the use of this tools, making this procedure fast to complete and improve, improving the viability of these prediction methods.

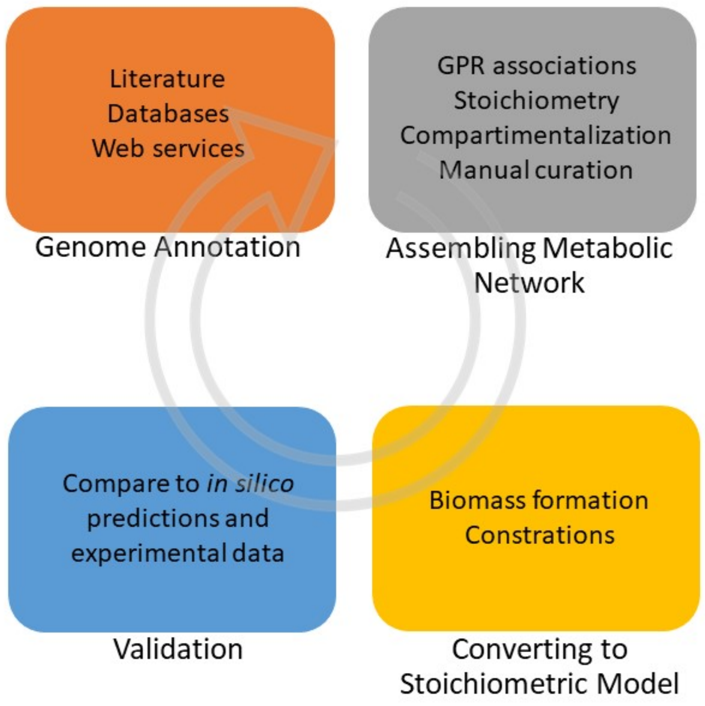


Figure 2.: Genome-scale metabolic model's reconstruction principals stages.

2.3 MERLIN

The *merlin* tool is an open-source, easy to use Java^{TM}application and in-house built in University of Minho. Compatible with Linux and Windows environments, it has an intuitive interface, allowing anyone to perform models' reconstruction for any organism with sequenced genome (8).

This tool accomplish the principal steps of reconstruction process (figure 3), including the genome's functional annotation. The reconstruction of the reactions' portfolio can also be achieved and tools to identify and annotate genes that encode transport protein can provided by merlin (8).

Moreover, through the use of external tools (which create reports that can be loaded into the merlin), merlin is able to perform the compartmentalization of the model, predicting the organelles' localization of the proteins encoded in the genome (8).

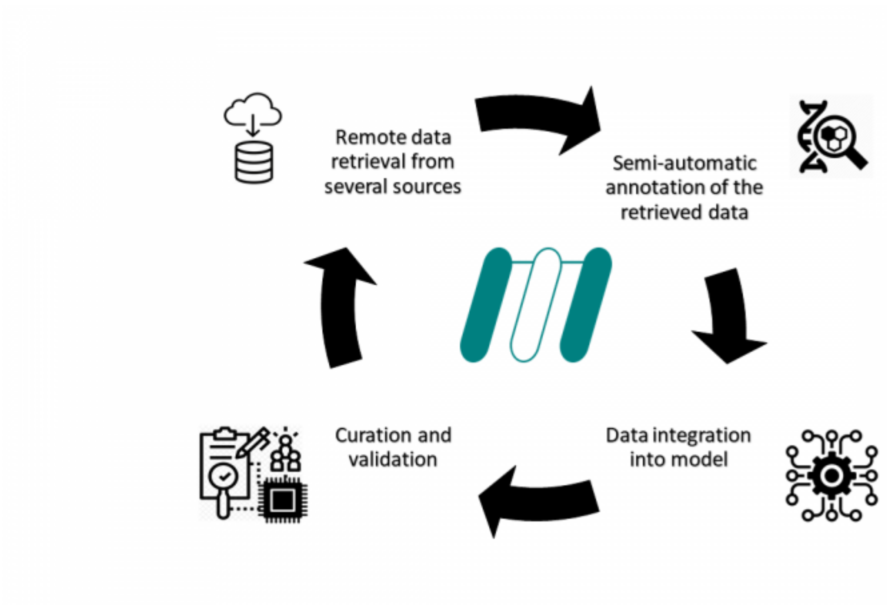

Remote data retrieval from several sources
Semi-automatic annotation of the retrieved data
Data integration into model
Curation and validation

Figure 3.: Main steps of reconstruction process, performed by *merlin*. Source: *merlin* (2018), metabolic models reconstruction using genome-scale information, [online] Available at: https://merlin-sysbio.org/index.php/Home [17-05-2019].

merlin also enables to share the computational models that represent biological processes, since it provides the genomic data conversion to scheme metabolic models reconstructions, exported in the *Systems Biology Markup Language (SBML)* (25) standard format (26).

To communicate with web services, a few Java^{TM}libraries, such as BioJava (27) and UniProtJAPI (28) are used by merlin (8).

For the GSM models' development, merlin gathers information from different databases and then stores the collected information in an internal relational database (8).

Currently, H2 (29) and MySQL^{TM}(30) are the two relational database engines supported by merlin to persist data (31).

H2 is an open-source and native Java^{TM}database. It can be used in to different ways: in server mode or embedded in the application. This engine can also be configured to run as an *in memory*/temporary database (the data will be maintained in the application memory and will not be stored on a hard disk) (32).

MySQL^{TM}is an open-source *Structured Query Language (SQL)* database management system, which uses the SQL language. SQL is a standard language for relational database management systems, allowing, through SQL statements, to execute tasks in the database (33).

Since merlin is an object-oriented application that stores data in relational databases, there is an incompatibility between a Java^{TM}application and a SQL database, as in an object-oriented application objects are represented by classes, whereas in an SQL database, data are represented by a tuples and queries are performed to return data. To overcome the SQL limitation and the conversion of stored data into merlin objects, it is essential to implement a persistence layer, which abstracts all this complexity, simplifying the work of the developer.

2.4 DATA PERSISTENCE

Data persistence can be described as the ability to store information in some place and be able to retrieve that information when it is needed. There are two types of data persistence: **persistent** (data that is stored, for example, on disk or cloud) and **transient** (data that exists in *Random-Access Memory (RAM)*). Most information used by an application is transient, or in other words, the data lifetime is equal to the process that instantiated it (34). Applications require persistent data, since information inputted by a user in a application must be preserved, even when the machine is turned off (34).

The data persistence can be implemented in object-oriented applications in different ways, such as (34):

- Through the **persistent object class** itself

In this case, is the persistent object class that defines the code for saving and loading objects in a data storage, such as a database. This is an example of direct mapping, which requires manual management of the data storage. However, this example presents some limitations, since it requires a high coupling between the class and the persistence mechanism.

- Through a **persistence layer**

Other classes are responsible for the mapping of persistent objects - this is a case of indirect mapping, in which for each class to be stored, a mapper is defined. The mapper is responsible for materialization (loading from database), dematerialization (saving in database) and caching of objects.

Advantageously, the persistence layer provides a greater degree of maintainability, encapsulating functionalities and thus reducing coupling within the application, being the most indicated way for data persistence (35).

As previously mentioned, there are several forms of persisting data, such as in databases. Object-oriented databases (36)(37)(38) are most appropriate for the persistence of objects manipulated by object-oriented applications, but due to some factors such as cost, it is necessary to find alternatives to perform persistence.

Since the relational storage model offers performance advantages and help reduce data redundancy, there is a tendency to store object-oriented application objects in relational databases. However, the difference between these two paradigms must be taken into account, so compatibility is required (35).

Object-Relational Mapping (ORM) is an approach that presents several strategies to reduce the difference between the two paradigms, using abstraction, which allows that a class can be mapped to a table in the relational database and attributes of the class to the table fields (explained in more detail below) (35).

Current databases provide a structured representation of persistent data, managed by the *Database Management System (DBMS)*, which is responsible for sharing data between multiple users and multiple applications (34).

Each time an object is created, the *Object-Oriented Database Management System (OODBMS)* assigns an *Object Unique Identifier (OID)* to each object to be stored, which is used primarily to establish relationships between persistent objects. Whenever an application references an object via its OID, the OODBMS converts this OID into a virtual memory address, so the object can be found quickly regardless of where it is stored (39). Since the OID is independent of the value/data of the object, the system will always access the correct object, regardless of any changes that may be made to the object. Once assigned to an object, the OID lasts the lifetime of the object (39).

In figure 4, the operation of data persistence can be seen. The relational database communicates with the object-oriented application, and vice versa, through an intermediary, which is the persistence layer.

Relational Database
Data persistence
Object-Oriented Application

Figure 4.: Operation of data persistence.

2.5 DATA ACCESS OBJECT

A *Data Access Object (DAO)* is an abstract interface used to retrieve data from the database in the form of model classes. It also provides an API able to handle with objects of structured data from different data storages, as example, files, web repositories or databases (40). In addition, since data exchange with the DBMS is centralized, DAOs give a single data access point, resulting in an object-oriented design application (40).

The main advantage of a DAO is the possibility to remove the database connection and management from the main code of applications with such design, since it abstracts and encapsulates the mechanisms of any database access, even providing an easier database migration between different storage engines (41).

The DAO use is convenient if (42):

1. Persistent storage will be accessed more than once or in concurrent tasks;
2. It is necessary to separate the client data resource interface from the database;
3. It is necessary/wanted to adapt an access API from a specific data resource to a generic client interface;
4. Different teams work in different parts of the application;

It can be inferred, therefore, that once the DAOs separates the requirements of a database from the application, its implementation allows the application development, without database limitations (43).

For instance, to create a simple DAO that handle with information stored in any storage engine, it is required to create methods to interact with it. The read and the save method are the most common examples of methods that interact with storage engines. However, the application that uses such DAO just interacts with the methods, so any limitations in the file structure/format are ignored.

Through the analysis of the figure 5, it is possible to perceive that, through DAOs, merlin abstracts the interaction with the database.

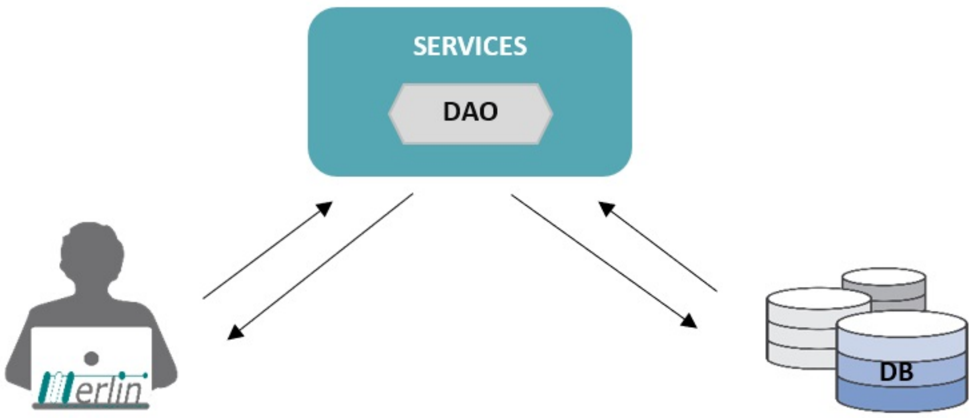


Figure 5.: Diagram representing *merlin*'s abstraction of the data persistence layer, through a database access object.

2.6 PREPARED STATEMENTS

As mentioned before, through SQL, the program can execute several tasks, however, the usage of SQL queries without parameterized commands leads to a security breach in the application, since it can be exploited by SQL injection which grants attackers the power to access and modify/delete information, without authorization, from a database (44).

To avoid injection vulnerabilities, the most efficient way to query a database with SQL is by adopting prepared statements with parameterized commands, where placeholders are used to separate literal values from the SQL query itself. Also, prepared statements have a static structure, preventing SQL injection attacks from changing their logical structure and manipulating the structure of the pre-compiled query (45)(46). Thus, the developer defines the query structure and using these placeholders is able to set the values to be passed to the final query. Afterwards, the application attaches the submitted values to the SQL query structure using a command interpreter. This procedure is named as database prepared statements or parameterized queries (47).

Previously, to perform its operations, merlin's APIs used prepared statements. Prepared statements can be defined as objects with a precompiled SQL queries that could be performed several times efficiently (48)(46)(47). Its execution comprises two stages: preparation and execution. Whereas in the first, the database server receive a statement template, in the execution stage, the server checks the syntax and its internal resources are initialized for use posteriorly. (49).

In Java^{TM}, when a prepared statement is created by the connection object, the prepared statement is precompiled. This one is sent to the database to be executed and the result set is sent back from the database (46).

However, prepared statements lacks some funcionalities, and so the application presents some limitations, such as (50):

1. Possibility for bugs, due to the need to develop code to manage database connections;
2. Inability to use distinct database engines, being necessary specific knowledge about the SQL syntax of each database engine;
3. Cache management taks up memory on the server. Since it occurs at database level, its access performance reduces.
4. The result of SQL are strings, obligating to parsers' development, to convert this type of data into model objects.

In order to outgrow the above described limitations, Hibernate *Java Persistence API (JPA)* queries can replace prepared statements.

2.7 OBJECT-RELATIONAL MAPPING

An ORM abstracts the communication between two paradigms previously mentioned: object-oriented databases and relational databases.

The ORM can be defined as the method that enables to convert data between object model and relational database, facilitating the work of the developer. (34).

According to Bauer and King, an ORM solution consists of four parts (34):

1. An API for performing basic operations, such as create, read, update and delete on persisted data.
2. A language/API for detail queries referring to classes and their properties.
3. A resource for detail mapping metadata.
4. A technique for the ORM implementation, to interact with transactional objects to perform "dirty checking, lazy association fetching, and other optimization functions".

The implementation of an ORM is complex but brings great benefits. With an ORM framework the application interacts with the ORM APIs, the domain model classes and is abstracted from the underlying SQL/*Java Database Connectivity (JDBC)*. So, its usage allows the manipulation of objects with data stored in a database, without the knowledge of the existent relationships between data objects within the database and provides a conceptual abstraction for mapping the application code to the underlying databases (51).

Other ORM advantage is that any data storage or storage APIs modification only occurs at the ORM level. This is possible given that ORM encapsulates changes in the data storage from the application itself (52). Besides that, an ORM implementation allows the application to be cheaper and less vendor-specific. Also, there is an increase in performance and the application becomes more able to handle changes to the underlying SQL schema (34).

Despite all this advantages, potential performance issues can be introduced, since developers may not be aware which part of source-code would result in a database access or if this access is or not efficient (51). Also, another disadvantage is that ORM frameworks are not lightweight tools.

Since Java^{TM}5, Java^{TM}has included a standard JPA that supports ORM (51). The JPA is developed to map Java data objects to database objects. This requires extensive support for new features in Java^{TM}languages, such as annotations and generics.

JPA is a standard Java API for persistence using a concept of ORM (53). It was introduced to replace Entity Beans, which have been discontinued, and to simplify the development of *Java Platform Enterprise Edition (JEE)* and *Java Platform Standard Edition (JSE)* applications that use data persistence. So, since with JPA the entities are *Plain Old Java Objects (POJO)*, nothing is required to make objects persistent, being just necessary to add annotations in the classes that represent the entities of the system and begin to persist or query objects (53).

A very relevant advantage of JPA is that it allows the development of an application with one or multiple databases integrated into the system, without any hard-coded queries which are only compatible with DBMSs of certain vendors (53).

There are some others JPA advantages, such as (53):

- **Standardization**

  JPA is one of the JEE standards, so any framework that complies with this standard follows the same architecture and provides the same API access, ensuring that applications developed using JPA require minor changes to be upgraded.

- **Support for container-level features**

  Unlike the limitations of other simpler frameworks, JPA features includes the support of big data sets, transactions, concurrency and other container level transactions are very important to enterprise applications.

- **Easy to use, easy to integrate**

  With the JPA framework, creating entities is so easy as creating Java^{TM}classes without any restrictions, using only annotations from *javax.persistence.Entity*. The JPA's framework and interfaces are also simple, without many special rules and patterns of software development.

- **Great capabilities of query**

  JPA has a unique Java^{TM}persistence query language, which can support large-scale updates and has SQL functions, such as *Join*, *Group By* and *Having*. In addition, it can also support a subquery.

- **Advanced features of object-oriented support**

  JPA supports object-oriented features such as inheritance across classes, allowing to maximize the use of object-oriented model of enterprise applications design.

Currently, there are some ORM frameworks available for Java^{TM}environments, for instance, MyBatis (54) and Hibernate (55).

2.8 OBJECT-RELATIONAL MAPPING TOOL HIBERNATE

Over time, developers have been reducing the efforts and time spent on joining an object-oriented application to a relational database.

Hibernate, created by Gavin King in 2001, has emerged to solve this issue by introducing an ORM approach (34)(56).

This framework is an Java^{TM}open-source ORM solution, created with the purpose of simplifying developer's work, converting Java^{TM}classes to database tables and Java^{TM}data types to SQL data types. Furthermore, it maps the fields/properties of domain classes to the database schema using either *eXtensible Markup Language (XML)* or annotations (57).

In short, Hibernate can be configured with a set of pre-finished configuration files, classes and interfaces that allow the creation of DAO layers. These layers are capable of abstracting functionalities of the database, making them available for computational Java^{TM}applications.

The Hibernate operation mode and mapping techniques can be seen in the figure 6.

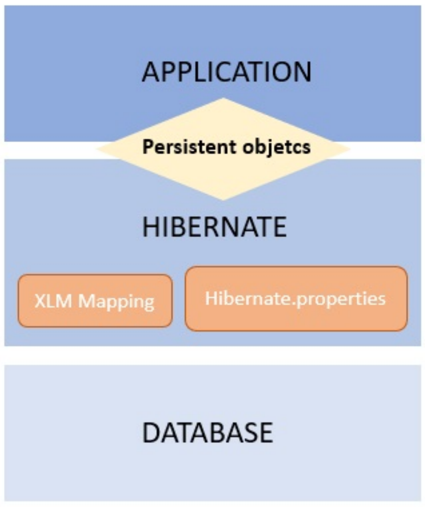


Figure 6.: Hibernate operation mode.

To map relational database objects, Hibernate adopts the JPA specification, responsible for standardizing the *Enterprise JavaBeans (EJB)* programming model and it requires three libraries that follow this specification: Hibernate Core, Hibernate Annotations and Hibernate EntityManager (58).

The *Hibernate Core* library contains an API and metadata in XML files to define the persistence layer. It has *Hibernate Query Language (HQL)* as its own database query language with proper interfaces for database management (58).

The *Hibernate Annotations* is a way to map the POJO objects of Java^{TM}into database tables, using special tags of JavaDoc type. The main advantage of this library is the reduction of lines of code required to carry out this mapping process (58).

The *Hibernate EntityManager* allows to access the database. It can be used to create and/or remove instances of entities through the primary key and allows to query over all entities (59).

Due to its features, Hibernate can overcome the limitations presented by SQL prepared statements. Also, can implement new functionalities for object-oriented applications (60)(61).

2.8.1 *Hibernate's new features*

With the release of Hibernate 5, several new features are available in order to facilitate common development tasks. Some of the most important are:

- **Transparent relational persistence for Java^{TM}**

  Unlike the prepared statements, where the developer has to write code to map database tables tuples to application objects, Hibernate maps Java classes to database tables and uses XML files to reduce the code requirements (61).

- **Query language support**

  Through Criteria API, Hibernate provides a query language adjustable to any type of database storage, such as MySQL^{TM}(62), Microsoft SQL Server (63), PostgreSQL (64) and Oracle (65). In addition, it can support native SQL statements (61).

- **Fetches management and performance improvement**

  Hibernate provides a query interface, that fetches the database query results in distinct batches and uses Hibernate's *ScrollableResults* implementation, allowing a faster visualization to the user, once data is paginated. Besides that, Hibernates cache is set to the application workspace, in order to increase fetches' performance (61).

In addition to these features, Hibernate also provides further ways of handling data and relational database management systems. JPA Criteria API, is one of the Hibernate's components, that allows the development of a criteria query object, enabling to apply filter rules and logical conditions (described in more detail below) (66).

2.8.2 *Hibernate's advantages*

Several benefits come with Hibernate's implementation, such as (56)(53):

- **Productivity**

  According to Christian Bauer and Gavin King (34), "the result, Hibernate, is a practical solution, emphasizing developer productivity and technical leadership", once Hibernate provides data query and retrieval facilities, significantly reduces code development time. Unlike with SQL and JDBC, where the developer needs to spend time with manual data handling, with Hibernate, the developer does not need to worry about creating, manipulating and changing tables present in the database, since this tool helps in this mapping and thus saves more time and work in developing the project (67)(55).

- **Maintenance**

  The system becomes more maintainable, given that with less code is easier to refactor. Thus, to facilitating understanding, fewer lines of code make the system easier to change.

- **Performance**

  Hibernate implements various techniques that allow to improve the performance of the specific persistence of a DBMS.

- **Vendor independence**

  Since the tool supports a number of different databases, this confers a certain level of portability on the application (vendor independence) (34).

2.8.3 *Hibernate's architecture*

Hibernate's architecture includes several interfaces required for database management that are distributed as follows (34)(56):

- *Session*, *Transaction* and *Query* interfaces are called by the application to perform the execute create, read, update and delete operations in the database;
- *Configuration* interface is used to configure Hibernate to define the database connection;
- *Interceptor*, *Lifecycle*, and *Validatable* callback interfaces are used to perform the verification and reaction events within the database management inside Hibernate framework;
- *UserType*, *CompositeUserType* and *IdentifierGenerator* are interfaces that allow extension of Hibernate's powerful mapping functionality.

In figure 7 it can be seen in general the architecture of Hibernate. In yellow and blue it can be seen the main interfaces of Hibernate.

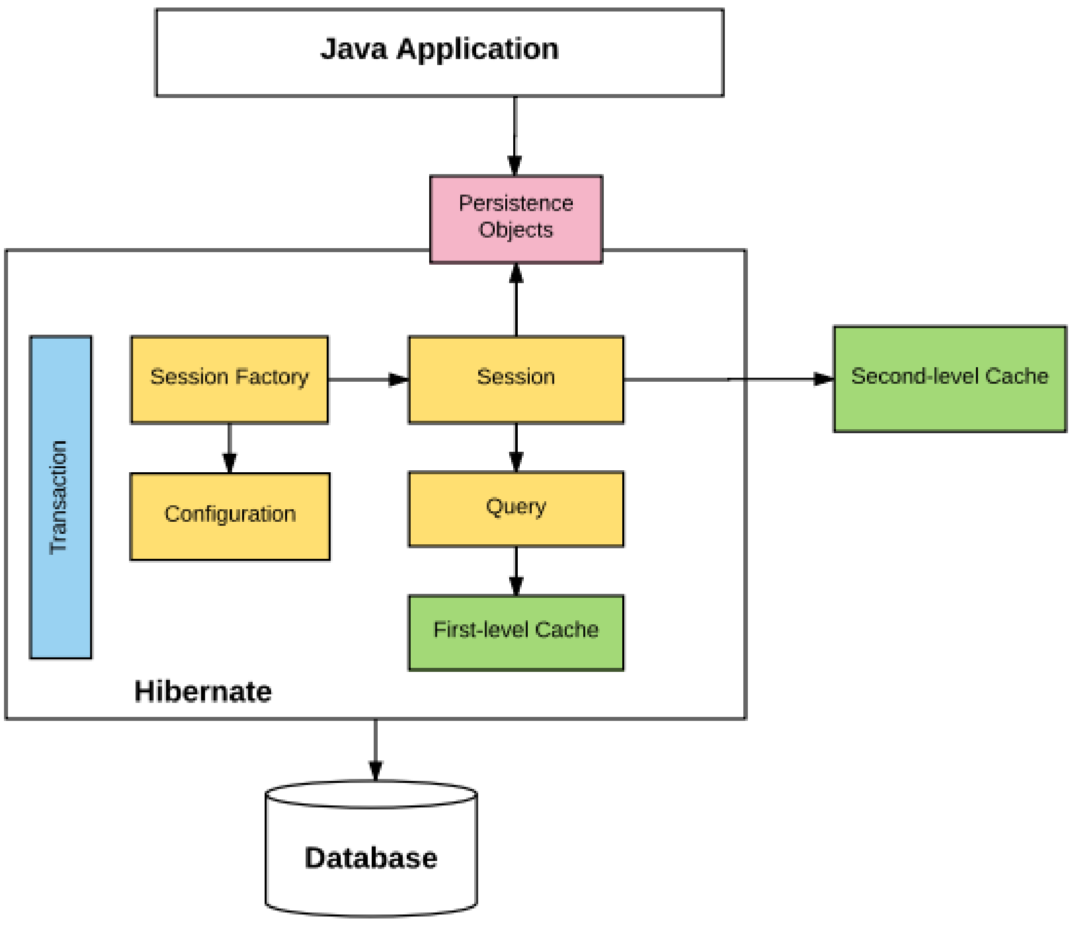


Figure 7.: Hibernate architecture. Source: HowToDoInjava (2016), Hibernate Architecture, [online] Available at: https://howtodoinjava.com/hibernate-tutorials/ [20-06-2019].

The interfaces described above are used in the business and persistence layers of the application. Through the analysis of the figure 8, it can be seen how these layers are structured, where the persistence layer is the only layer directly connected with the database in a layered architecture.

Hibernate framework resorts to some Java APIs, for instance, JDBC, Java Transaction API and *Java Naming and Directory Interface (JNDI)*. Whereas JDBC allows almost any database with a JDBC driver to be supported by Hibernate, JNDI and *Java Transaction API (JTA)* allow Hibernate to be integrated with JEE application servers (34).

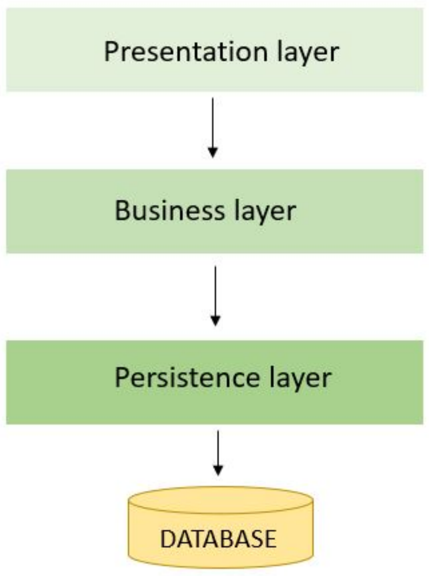


Figure 8.: Layered architecture.

2.9 CRITERIA API

The Criteria API is based on the abstract schema of persistent entities, their relationships and embedded objects (68).

To retrieve or define the data in the database using entity objects, criteria query objects are required. To build these dynamically, JPA Criteria API can be applied (69)(57).

Data fetching is simplified using Criteria API, differently from HQL and native SQL query languages that require a specific syntax. This API uses only objects to persist, extract and modify database entities (70). This feature confers advantages, since data manipulation does not require any hard-coded SQL statements and its programmatic behavior offers compile-time syntax checking, thus allowing to detect errors earlier (69).

The Criteria API includes the *Query By Example (QBE)* functionality, through which an object can provide the properties able to be modified, rather than having to define the components of a query step-by-step. In addition, it also includes projection and aggregation methods (70).

Criteria queries are a secure way of expressing a query, since interfaces and classes are used as representation of various structural parts of the query. These are basically an object graph, where each part represents an increase of a smaller part of the query. To execute a query, the first step is to create this graph. The *CriteriaBuilder* interface is a factory of objects that represent all the individual parts of the criteria (71). Then, it is expected to obtain the *CriteriaQuery*, through the *createQuery()* method, that requires an input type that can be an entity, an integer, a boolean or any other object (71).

Several database operations can be processed using the *CriteriaQuery()* where one or multiple entities attributes can be passed as inputs.

It can be inferred that the Criteria API contains features that make it easier to construct queries in Hibernate. Unlike HQL, which is harder to learn, the Criteria API is simpler and cleaner.

2.10 AIBENCH

Due to the fast development of several specialized applications with advanced features in the biomedical field, it became necessary to develop sophisticated user interfaces, since according to Glez-Penã et al., the existing frameworks do not provide support to dynamic graphical user interface, customization of default behaviour and application aspect, design of a clear application workflow, automatic script construction for supporting workflow repeatability, update service for automatically deploying software upgrades and automatic generation of technical documentation (72). Thus, the open-source Java desktop application framework *Artificial Intelligent workBench (AIBench)* emerged. This application aims to improve the quality and productivity in the development of applications directed to biomedical field and clinical research.

In order to allow a fast development of the application, three concepts are present in all AIBench applications: operations, datatypes and views (72).

The application layer is at the top of the architecture and include the application specific code, that is, the operations, the data-types and the views, allowing it to be reusable in more than one final application, by reusing libraries from past projects or third-party software (72).

3

PROPOSAL SOLUTION

"Nothing is impossible in computing, absolutely anything is possible.
It is necessary to have the knowledge, time, patience to research, investigate, persevere and motivate to progress and reach our goals."
Giovanni Pugliese

All paths and resources that make it possible, should be studied to achieve a goal.

An analysis to the data and business layers is an essential first step, to understand how it was developed and organized. This analysis allows choosing the paths to be followed and the resources to be used.

As mentioned before, the main goal of this thesis was to implement an ORM in merlin's framework, to remove the dependencies of MySQL^{TM}and H2 databases from merlin's core. Hence, all tools and resources presented in previous chapters, will be used to facilitate this process.

Thus, for this thesis, after the referred initial analysis, it was proposed to implement Hibernate in the merlin framework, according to the intended purpose.

To make this implementation possible, an analysis and study of all existing code and database model was made. A summary of this study is presented below.

3.1 MERLIN ARCHITECTURE: PRE-IMPLEMENTATION

3.1.1 *Application Workspace Structure*

The previous analysis, allowed determining that the software was divided into different projects. Each project had its own dependencies, as shown in figure 9.

The "core" project, has a misleading name as it is the main project, which depends on the "biomass", "gpr" and "triage" projects. However, these also have their dependencies, and so on, so the "core" project becomes the project that depends on all other existing projects.

The "utilities" project is the project that does not depend on any other project and it can be considered an auxiliary project.

This architecture was changed during the development of the general project and will be shown later.

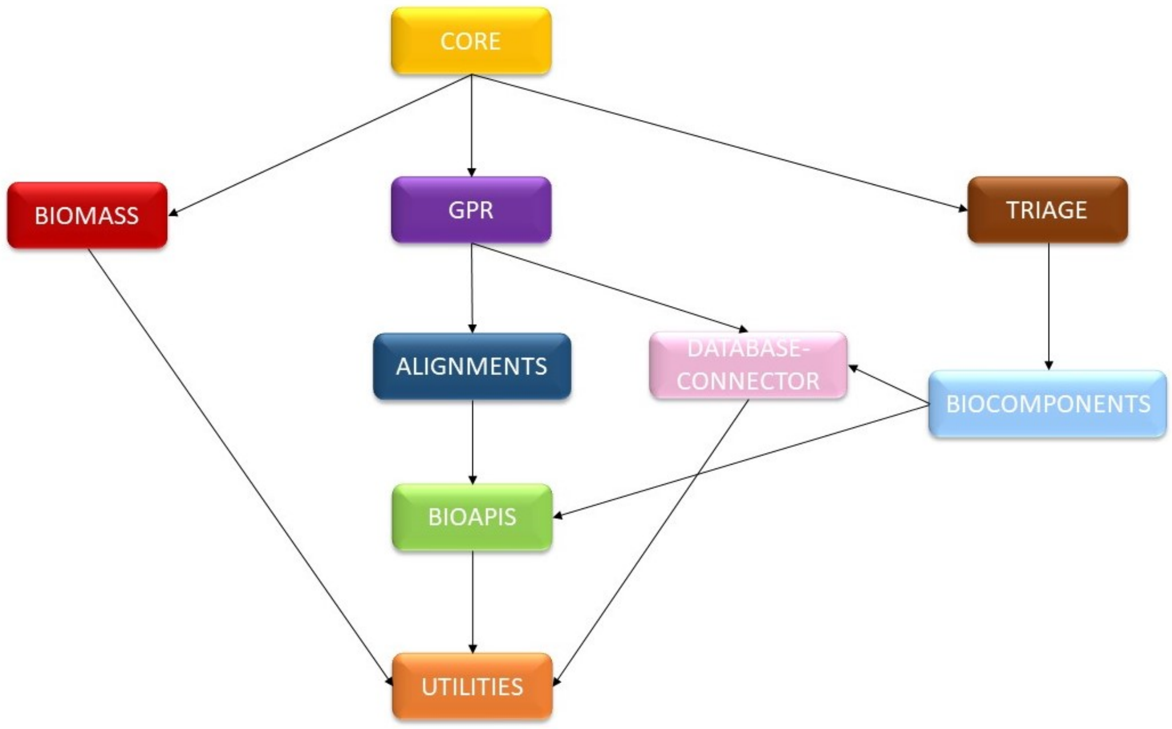


Figure 9.: Previous project architecture.

3.1.2 *Data and Business Layers Code*

The code referring to the data and business layers was coupled and organized in different classes, corresponding to different database APIs, which include business and database access logic. SQL queries were spread all through the code, embedded in JDBC statement objects.

3.1.3 *Database APIs*

The different database APIs were present under the "database-connector" project and were organized as follows:

- The *"ProjectAPI"*, containing all the queries and business logic related with a merlin project. This API contained all methods that could be related to more than one API.

- The *"ModelAPI"*, that includes all the queries and business logic related to the models. This API managed 52 database tables, incluing:
  - "model_compound", which contains the compounds name, *International Chemical Identifier (Inchi Key)*, formula, molecular weight, charge, *Simplified Molecular Input Line Entry System (SMILES)*
  - "model_enzyme" containing information concerning to enzymes, as *Enzyme Commission (EC)* number, source, GPR status and optimum *Potential of Hydrogen (Ph)* (the Ph value where the enzyme is most active.
  - "model_gene", where it can be found all the information regarding the gene in study, as name, locus tag, transcription direction, origin and the sequence *identifier (ID)* of the protein that it encodes.
  - "model_compartment", referents to the model location;
  - "model_gene_has_compartment" that links the two previous tables, providing the primary location and score.
  - "model_pathway", where are all KEGG pathways, and that relates to the "model_enzyme" table through the "model_pathway_has_enzyme" table;
  - "model_protein", where the protein name, class and molecular weight are.
  - "model_reaction", containing all the reaction information, as name, equation, reversibility, lower bound, upper bound and source.

- The *"HomologyAPI"*, that includes all the queries and business logic for the operations needed to create and edit enzymes annotation.

  "HomologyAPI" contained SQL queries allow us to identify genes homologous to the gene under study. It is possible to find homologous proteins sequences through Blast, according to e-value, and then, assuming these proteins have similar functions, assigning a function. It is also possible through Interpro, collect the functional analysis of proteins, classifying them into families and predicting its domains.

  This API managed 30 *merlin* database tables, including:
  - "enzymes_annotation_ecnumber", which contains all ecnumber;
  - "enzymes_annotation_fastasequence" containing FASTA sequences;
  - "enzymes_annotation_genehomology" containing locus tag, protein ID, gene name and *The Universal Protein Resource (UniProt)* EC number;
  - "enzymes_annotation_homologues", which contains genes locus tag, all the proteins they encode and calculated molecular weight, in order to determine stoichiometry in chemical reactions catalyzed by these protein.

  - "enzymes_annotation_genehomology_has_homologues" links the two previous tables, giving us homologous proteins and their e-value values.
  - "enzymes_annotation_organism", containing information concerning to the organism, its taxonomy and taxonomic rank.

- The *"CompartmentsAPI"*, that includes all the queries related with compartments' annotation.

  "CompartmentsAPI" managed the compartmentalization related tables. Through this API it was possible to retrieve/insert/update/delete information concerning cellular location relative to a target organism and reactions that occur in a given compartment. It was responsible for three tables:

  - the "compartments_annotation_compartments", which includes a list of all possible compartments and their abbreviations;
  - "compartments_annotation_psort_reports", which includes the gene tag locus;
  - "compartments_annotation_psort_reports" that links the two previous tables.

These database APIs receive arguments and a statement object. This statement object is created from the connection with the database, executing a database query, which could either result in database changes, or returned requested information.

As stated before, the different database APIs were created under the database-connector project, that contained all the interactions with merlin's internal database.

3.2 DATA AND BUSINESS LAYERS IMPLEMENTATION PROPOSAL

It was necessary to reformulate virtually the data and business layers code to reach the main objective of this thesis.

After the previous study, it was decided to restructure merlin's architecture. A new architecture was designed, in which all the layers are now independent, as shown in image 24, and in more detail, with all the new packages included, in figure 25. The difference between the previous architecture and the improved architecture is explained in the next chapter.

This new architecture involves using interfaces, creating an abstraction layer between the various application layers (*Graphical User Interface (GUI)*, business logic layer and data layer).

Thus, the proposed architecture aimed at making the data layer more generic and adaptable to various types of databases. Similarly, by making the business logic layer independent, the database can be changed without having implications in the data layer.

These new implementations allowed to remove existing dependencies between the application and the database.

In addition to these implementations, to allow compatibility with the Hibernate implementation, there was a need to change the initial merlin internal database schema. Other changes, which will be described in the next chapter, have also been designed to improve database performance.

Notice that there is no single solution to design the architecture of an application. Hence, any attempt is a learning opportunity. This implementation was one solution in several options.

The entire reformulation process is detailed in the next chapter.

4 PRACTICAL APPROACH

In this chapter the entire implementation process is described and graphically presented.

Due to the complexity of the overall project, only a part will be presented. Thus, the entire implementation will be demonstrated using as example the "gene" table of the database. This table contains all the information regarding the genes present in the merlin database. The process that will be exemplified for this table was the same for the other 90 tables in the database, however, with the exception of generic methods, the only difference is that each table has its own specific methods implemented.

As it has been mentioned throughout this dissertation, the main objective of this project is to remove the dependencies of the MySQL™and H2 databases from merlin's core. For this, the data layer and the business logic layer were restructured (both were previously in a single layer), making them now independent of each other.

The main phases of this implementation are presented next.

4.1 CHOSEN TOOLS

This thesis practical approach was implemented using only open-source tools.

The *Integrated Development Environment (IDE)* used was Eclipse, version Oxygen.2 Release (4.7.2). This tool offers several plug-ins that add several functionalities, easing the development of the code.

This work was developed using Hibernate version 5.2.15 and MySQL™JDBC driver version 5.1.34.

Hibernate Tools were used to help in the initial project setup, facilitating project generation.

Unit tests were implemented using JUnit version 4.12.

4.2 IMPLEMENTATION

4.2.1 *Database cloning to local environment*

The first step of this practical work was the cloning of the production database to the local development environment, to facilitate the connection and not change data in the production database during the tests.

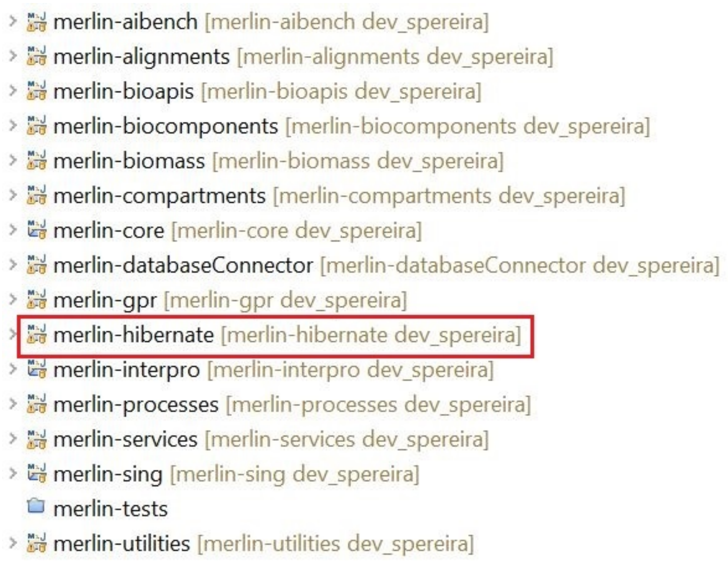
MySQL Community Server™version 5.7.19 was installed locally as database server and MySQL WorkBench™version 6.3.9 was installed to interact with the database.

Then in the MySQL WorkBench™, a connection to the local MySQL Server™was created. All necessary SQL instructions were exported through MySQL WorkBench™to have a copy of the production database in local environment.

4.2.2 *Eclipse Project Initial Setup*

The new project architecture was intended to not be highly coupled. With this propose, a Maven project was created, as Maven allows separating the application in several projects having their own life cycle. Nevertheless, a project can become very complex, so this complexity is hierarchized in sub-directories.

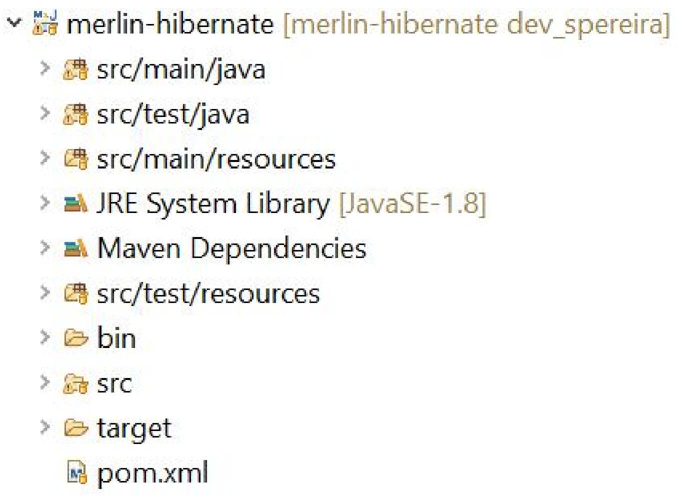
For this thesis, within the existing merlin Eclipse workspace (10), a new Maven project called ”merlin-hibernate” (figure 11) was created for new merlin module.



```
merlin-aibench [merlin-aibench dev_spereira]
merlin-alignments [merlin-alignments dev_spereira]
merlin-bioapis [merlin-bioapis dev_spereira]
merlin-biocomponents [merlin-biocomponents dev_spereira]
merlin-biomass [merlin-biomass dev_spereira]
merlin-compartments [merlin-compartments dev_spereira]
merlin-core [merlin-core dev_spereira]
merlin-databaseConnector [merlin-databaseConnector dev_spereira]
merlin-gpr [merlin-gpr dev_spereira]
merlin-hibernate [merlin-hibernate dev_spereira]
merlin-interpro [merlin-interpro dev_spereira]
merlin-processes [merlin-processes dev_spereira]
merlin-services [merlin-services dev_spereira]
merlin-sing [merlin-sing dev_spereira]
merlin-tests
merlin-utilities [merlin-utilities dev_spereira]
```

Figure 10.: *merlin*’s Maven projects.

To create the ”merlin-hibernate” Maven project, where all the code was developed, it was necessary to install ”Hibernate Tools” on Eclipse. These tools facilitates the project generation, preparing the development environment with all the tools needed for project development, such as JUnit for testing.



```
merlin-hibernate [merlin-hibernate dev_spereira]
  src/main/java
  src/test/java
  src/main/resources
  JRE System Library [JavaSE-1.8]
  Maven Dependencies
  src/test/resources
  bin
  src
  target
  pom.xml
```

Figure 11.: merlin Maven project module ”merlin-hibernate”.

This project contains a pom.xml configuration file, which is a XML based document were all project properties are described, as the version and dependencies. So, the dependency of Hibernate was referenced here.

It also has a *src* folder, containing all the source-code of the project. The *src* folder typically contains a *main* folder, where all the source-code of the project is, and a *test* folder, where the unit tests code are.

Finally, this project has the *bin* folder, where Maven stores the auxiliary files for the project build.

The project connection to the database was made via ”Database Development”, an graphical interface for database administration available in Eclipse (figure 12).

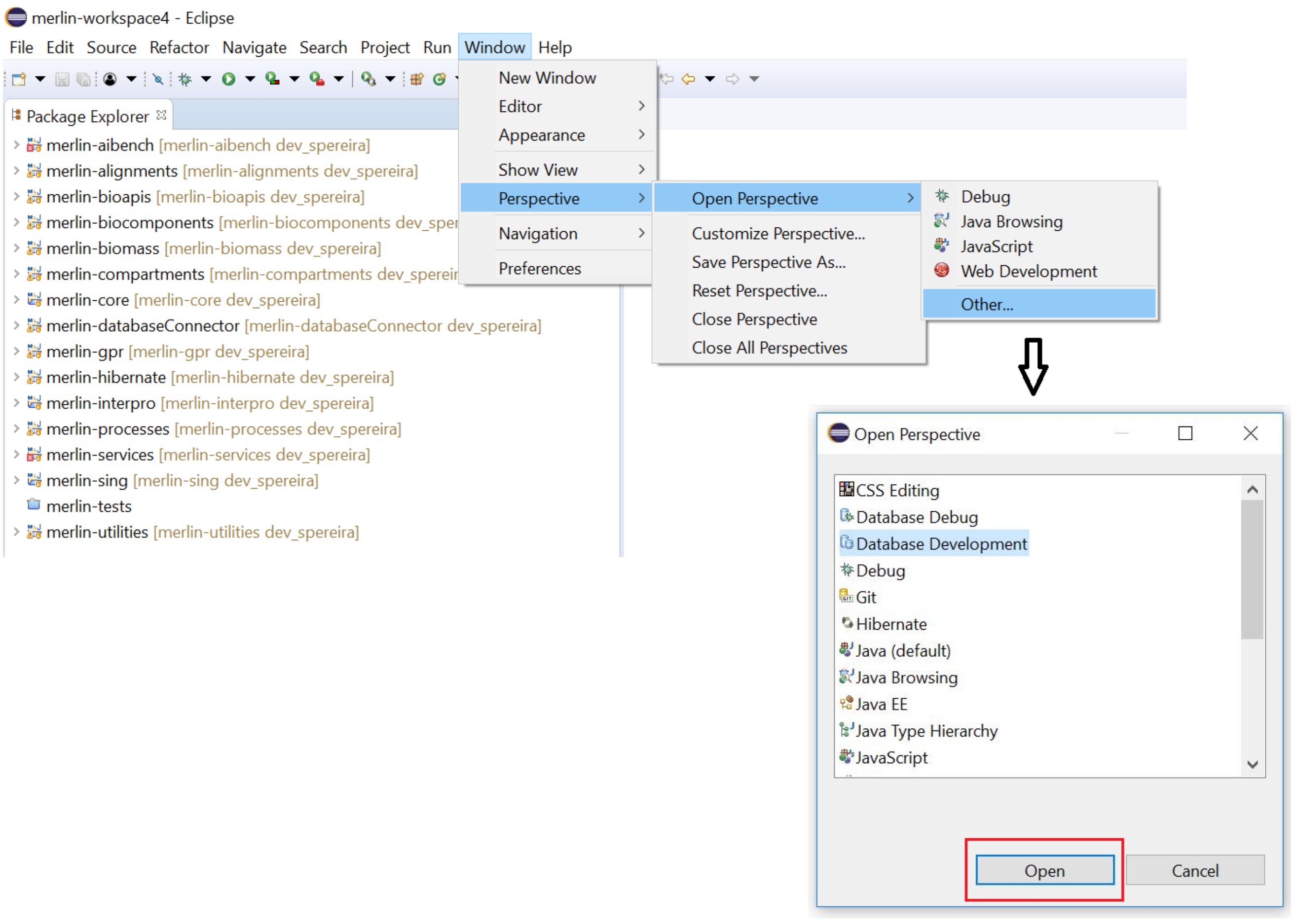

merlin-workspace4 - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
merlin-aibench [merlin-aibench dev_spereira]
merlin-alignments [merlin-alignments dev_spereira]
merlin-bioapis [merlin-bioapis dev_spereira]
merlin-biocomponents [merlin-biocomponents dev_sper
merlin-biomass [merlin-biomass dev_spereira]
merlin-compartments [merlin-compartments dev_spereir
merlin-core [merlin-core dev_spereira]
merlin-databaseConnector [merlin-databaseConnector dev_spereira]
merlin-gpr [merlin-gpr dev_spereira]
merlin-hibernate [merlin-hibernate dev_spereira]
merlin-interpro [merlin-interpro dev_spereira]
merlin-processes [merlin-processes dev_spereira]
merlin-services [merlin-services dev_spereira]
merlin-sing [merlin-sing dev_spereira]
merlin-tests
merlin-utilities [merlin-utilities dev_spereira]
New Window
Editor
Appearance
Show View
Perspective
Navigation
Preferences
Open Perspective
Customize Perspective...
Save Perspective As...
Reset Perspective...
Close Perspective
Close All Perspectives
Debug
Java Browsing
JavaScript
Web Development
Other...
Open Perspective
CSS Editing
Database Debug
Database Development
Debug
Git
Hibernate
Java (default)
Java Browsing
Java EE
Java Type Hierarchy
JavaScript
Open
Cancel

Figure 12.: Database connection setup: Open database development perspective.

In "Database Development", a new connection to the database was created (figure 13). The type of the database was set to MySQL^{TM}(figure 14), and then the JDBC connector configured.

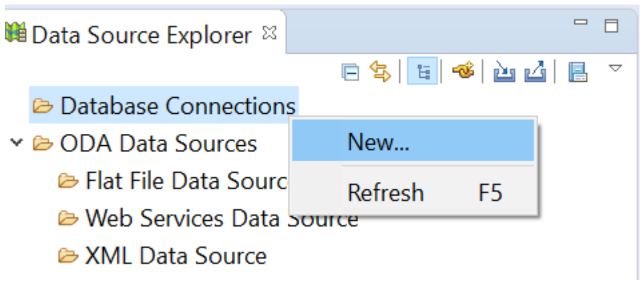
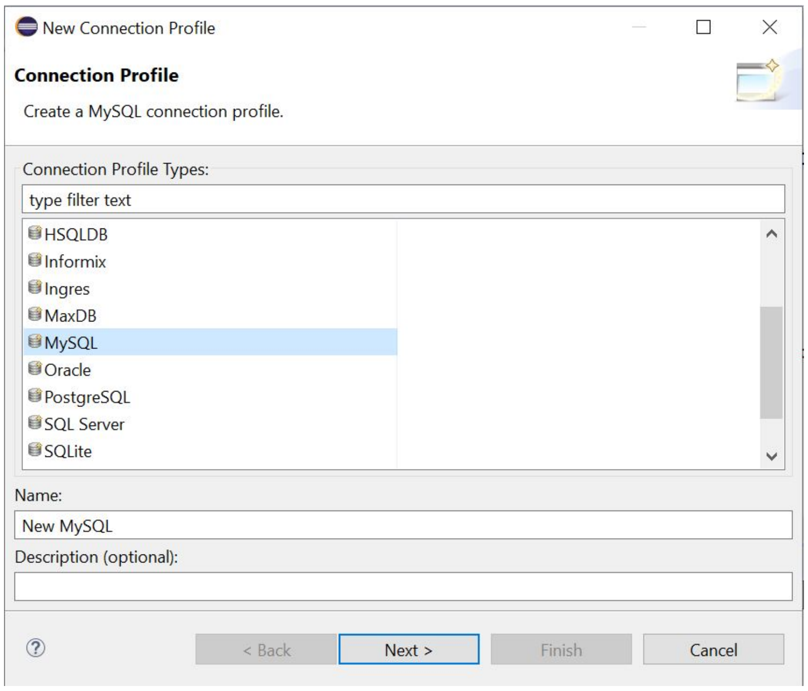


Figure 13.: Database connection setup: New database connection.

All the *Java ARchive (JAR)* were added and the connection details, as "host", "port number" and "user name" were fulfilled.


New Connection Profile

Connection Profile

Create a MySQL connection profile.

Connection Profile Types:

type filter text

- HSQLDB
- Informix
- Ingres
- MaxDB
- MySQL
- Oracle
- PostgreSQL
- SQL Server
- SQLite

Name:

New MySQL

Description (optional):

Figure 14.: Database connection setup: Database connection profile selection.

Once the connection was tested, the database structure could be seen in ”Data Source Explorer”.

4.2.3 *Java Entities Generation*

A new Maven project, named *”merlin-hibernate”* was created in the previous step. The Hibernate *group Id* and *artifact Id* were now added to the archetype parameters.

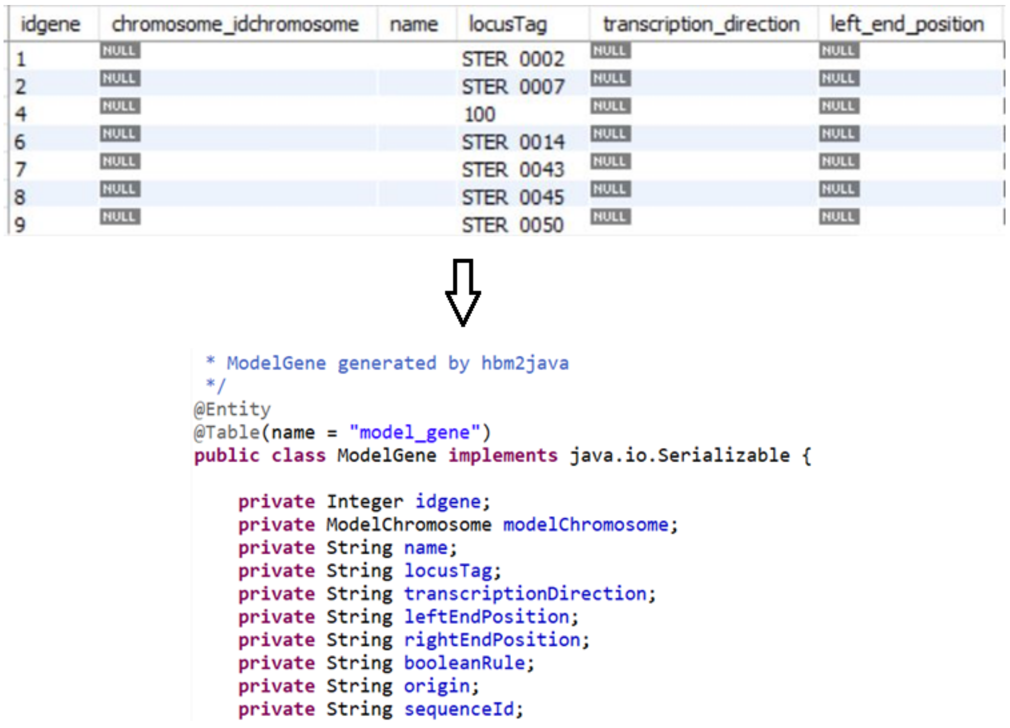
In the pom.xml file of this created project, the dependencies of Hibernate and the JDBC driver for MySQL^{TM}have been declared.

Then, another perspective was open to configure Hibernate and generate configuration file, using ”Hibernate Tools”.

In the configuration window, the previously created project was selected. A Hibernate configuration file, named ”hibernate.cfg.xml” was created in the *src/main/java* folder of the project.

Next, the ”Hibernate Code Generation Configurations” option were used to read the database structure and generate the entities classes. Thereby, through reverse engineering an ”hibernate.reveng.xml” was created. The schema to be included (all existing tables in the database were included) was specified in this reverse engineering file. Finally, the 141 entities classes, presented in appendices A, B and C, were generated.

The ”gene” table and the corresponding generated java class are depicted in figure 15. Each column of the database is represented as a private property of the class.



idgene	chromosome_idchromosome	name	locusTag	transcription_direction	left_end_position
1	NULL		STER 0002	NULL	NULL
2	NULL		STER 0007	NULL	NULL
4	NULL		100	NULL	NULL
6	NULL		STER 0014	NULL	NULL
7	NULL		STER 0043	NULL	NULL
8	NULL		STER 0045	NULL	NULL
9	NULL		STER 0050	NULL	NULL

```
 * ModelGene generated by hbm2java
 */
@Entity
@Table(name = "model_gene")
public class ModelGene implements java.io.Serializable {

    private Integer idgene;
    private ModelChromosome modelChromosome;
    private String name;
    private String locusTag;
    private String transcriptionDirection;
    private String leftEndPosition;
    private String rightEndPosition;
    private String booleanRule;
    private String origin;
    private String sequenceId;
```

Figure 15.: Gene table mapped to Java class.

4.2.4 *Database access layer implementation*

It was necessary to create DAOs classes to access the database. A package, named ”dao” was created under the ”merlin-hibernate” project. This package encompassed all the DAOs, interfaces and services.

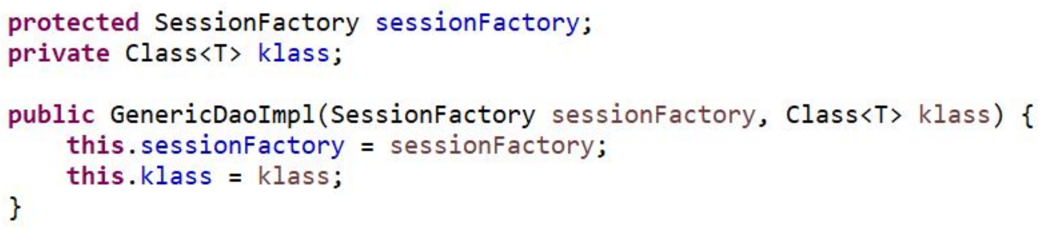
Initially, a generic interface was created, containing common methods to all other DAOs, such as *insert()* and *update()* methods.

This class contains one property of type *SessionFactory* and another private property of type *Class* (figure 16).

The *Class* type property is used as the database entity which the implemented DAO is managing, that represents the database table where the DAO will access. For example, the ”modelGene” DAO has in the *Class* property, the *ModelGene* entity class.

Regarding the *SessionFactory* property, it is created by a *Configuration* object, matching Hibernate configuration information. It then uses this information to generate an appropriate *SessionFactory* instance.

There is usually only one instance of one *SessionFactory* per main application thread.



```
protected SessionFactory sessionFactory;
private Class<T> klass;

public GenericDaoImpl(SessionFactory sessionFactory, Class<T> klass) {
    this.sessionFactory = sessionFactory;
    this.klass = klass;
}
```

Figure 16.: Generic DAO properties.

Using *SessionFactory* is possible to fetch the current session and through it create a criteria builder enabeling the usage of criteria API.

Thus, with these properties mentioned above, it was possible to create generic database access methods that could be transversal to all DAOs.

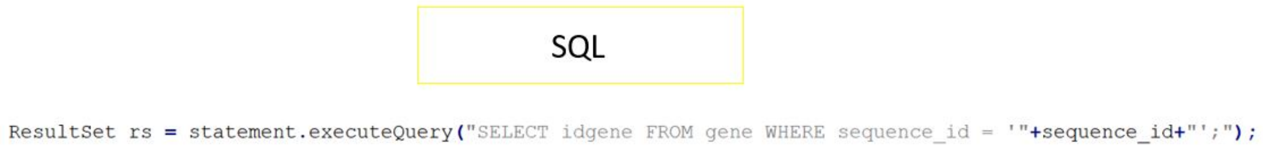
Then, the generic DAO object was implemented using the generic interface, which contains the standard methods of database interactions (select/save/update/delete database entities) using the *session* object.

During the integration of Hibernate framework, 91 DAO were implemented with their interfaces repectivley. The specific methods created in each DAO were created following the existing SQL queries in the APIs. Each DAO corresponds to a database table, so all methods to interact with a given database table are exclusively in their corresponding DAO.

All the methods were implemented using the JPA Criteria API, which integrates with Hibernate framework, to abstract the type of database, that is, making all the code developed adaptable to any type of database.

In figure 17 it can be seen how, through SQL queries, the access to database were previously made. This is a proper syntax of MySQL™databases, so for another type of database, such as H2, the syntax of the query can be different.

Also, in this case, the queries are not dynamic and the developer needs to create them dynamically to support filters. In this query a request is made to the "gene" table of the database, to obtain the ID of the gene whose sequence ID is inserted as a filter.

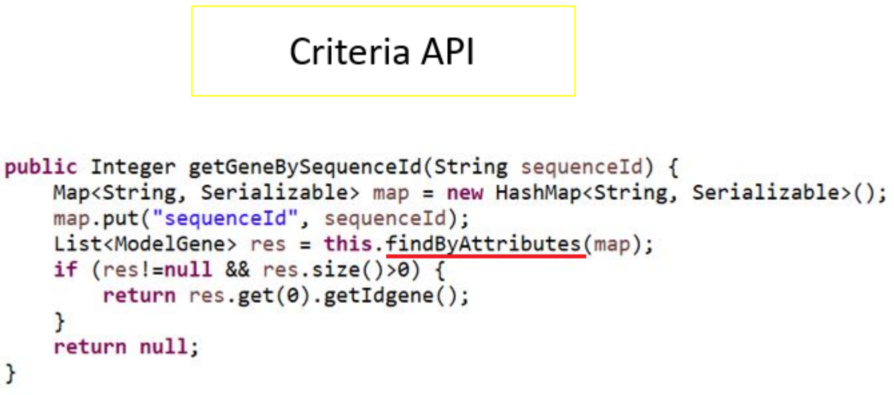


```
ResultSet rs = statement.executeQuery("SELECT idgene FROM gene WHERE sequence_id = '"+sequence_id+"';");
```

Figure 17.: Example of previous implemented SQL Query.

On the other hand, a query to access to the database using only the Criteria API is shown in figure 18. This query has exactly the same purpose as the previous SQL query. However, all filters are now added to a dictionary (in this case just a filter). Then, the entire

”ModelGene” properties are returned, that is, the entire table row fields, whose sequence ID is indicated by the *findByAttributes()* method. Finally, if the previous result is different from null or empty (thus indicating that there was matches), only the gene ID is returned.



```
public Integer getGeneBySequenceId(String sequenceId) {
    Map<String, Serializable> map = new HashMap<String, Serializable>();
    map.put("sequenceId", sequenceId);
    List<ModelGene> res = this.findByAttributes(map);
    if (res!=null && res.size()>0) {
        return res.get(0).getIdgene();
    }
    return null;
}
```

Figure 18.: Example of a current Criteria query.

The *findByAttributes()* method that finds the results by attributes can be seen more specifically in figure 19.

This method is transversal to all DAOs and is found in generic DAO. First, through the *SessionFactory*, the *CurrentSession* was fetched to get a *CriteriaBuilder*. It allows constructing criteria queries, compound selections, predicates, expressions and orderings.

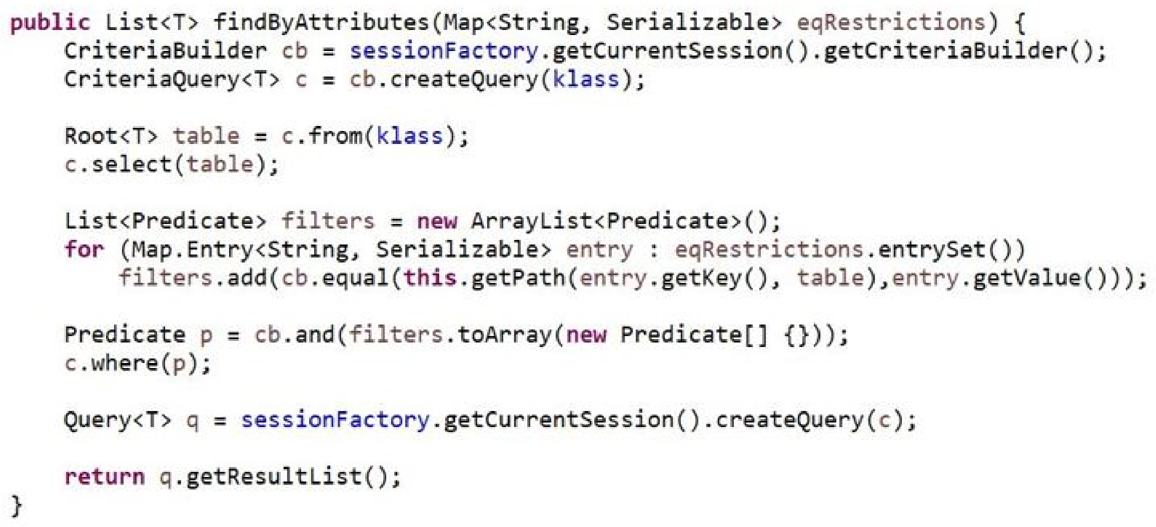
The *createQuery()* method was executed, specifying the class, to create the query. This method returns a *CriteriaQuery* capable of selecting, grouping and sorting query information.

Then, the table of interest was selected and predicates added to a list, allowing the query to be dynamic depending on the parameters that were passed.

The *CriteriaBuilder* method *and()* constructs a logical expression by joining all predicates with the logical connector *and*. Then, this filter list is set in *CriteriaQuery*. Finally, the result list is returned and only contains results that match all filters.

As this is an attribute search method, equality search was defined. Thus, the dictionary key corresponds to the field to filter and the value to its value to filter.

Lastly, it executes the query and returns the results.



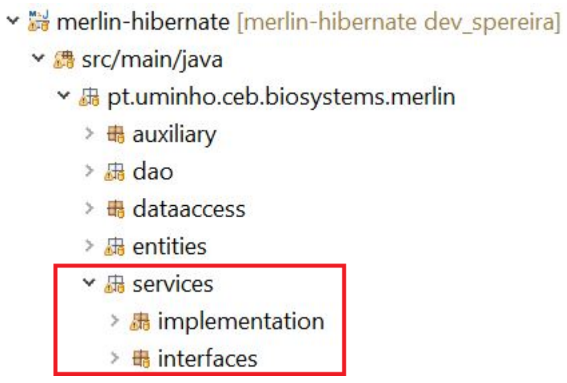
```
public List<T> findByAttributes(Map<String, Serializable> eqRestrictions) {
    CriteriaBuilder cb = sessionFactory.getCurrentSession().getCriteriaBuilder();
    CriteriaQuery<T> c = cb.createQuery(klass);

    Root<T> table = c.from(klass);
    c.select(table);

    List<Predicate> filters = new ArrayList<Predicate>();
    for (Map.Entry<String, Serializable> entry : eqRestrictions.entrySet())
        filters.add(cb.equal(this.getPath(entry.getKey(), table),entry.getValue()));

    Predicate p = cb.and(filters.toArray(new Predicate[] {}));
    c.where(p);

    Query<T> q = sessionFactory.getCurrentSession().createQuery(c);

    return q.getResultList();
}
```

Figure 19.: Search by attributes method.

4.2.5 *Business logic layer implementation*

All the APIs were reimplemented in services, that access to the database using the implemented DAOs, keeping all the business logic previously implemented.

For this purpose, a new package under the ”merlin-hibernate” project was created and named ”services”. This package included an ”implementation” and ”interfaces” subpackages (figure 20).

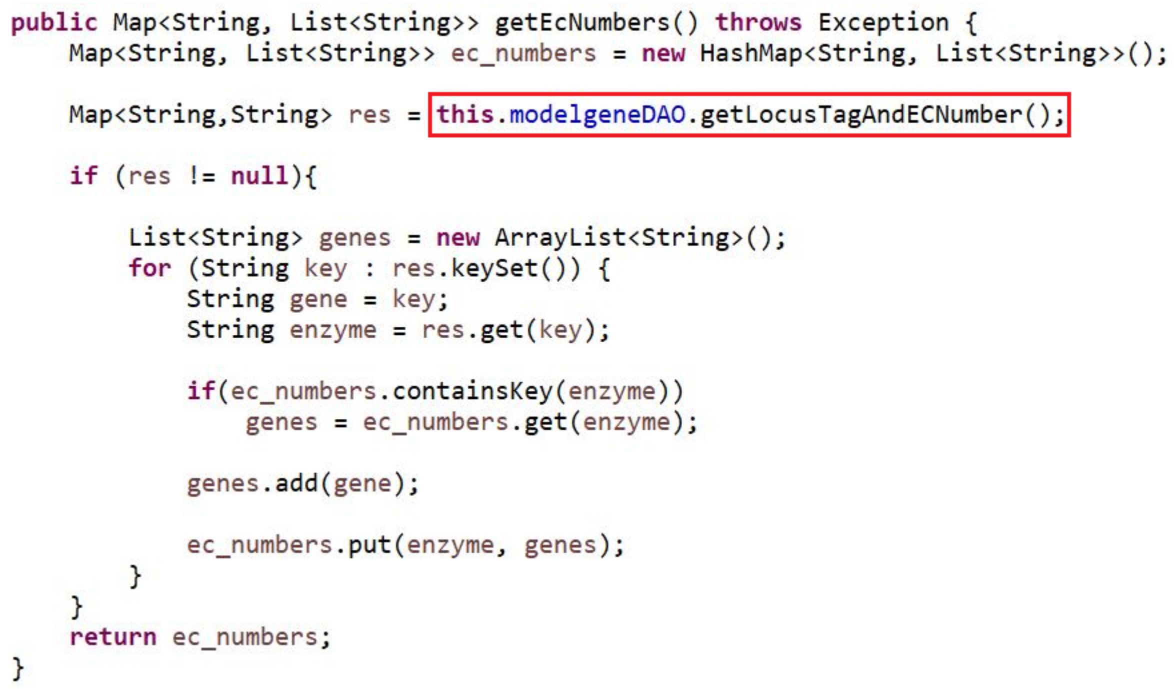


```
merlin-hibernate [merlin-hibernate dev_spereira]
  src/main/java
    pt.uminho.ceb.biosystems.merlin
      auxiliary
      dao
      dataaccess
      entities
      services
        implementation
        interfaces
```

Figure 20.: Services package structure.

In ”interfaces” subpackage, a service was created for each type of DAO, in a total of 19 interfaces. This means that, for example, all services referring to ”annotation” tables will be in the ”IAnnotationService”.

In the ”implementation” subpackage all the services are created, implementing an interface. In these services implementation all the business logic existing in the APIs was implemented, replacing the database access by the respective DAO and corresponding method.



```java
public Map<String, List<String>> getEcNumbers() throws Exception {
    Map<String, List<String>> ec_numbers = new HashMap<String, List<String>>();

    Map<String,String> res = this.modelgeneDAO.getLocusTagAndECNumber();

    if (res != null){

        List<String> genes = new ArrayList<String>();
        for (String key : res.keySet()) {
            String gene = key;
            String enzyme = res.get(key);

            if(ec_numbers.containsKey(enzyme))
                genes = ec_numbers.get(enzyme);

            genes.add(gene);

            ec_numbers.put(enzyme, genes);
        }
    }
    return ec_numbers;
}
```

Figure 21.: ModelGene service *getEcNumbers()* implementation.

For example, figure 21 shows a simple example in terms of the logic of a service implementation. It can be seen that this service uses the DAO ”modelgeneDAO” (in blue) to fetch all locus tag and EC number from the database. Then it aggregates the result by locus tag and return it.

So, whenever a request is made, these services will be in charge of processing the data and, through the DAOs, access the database, and return the expected data to the application.

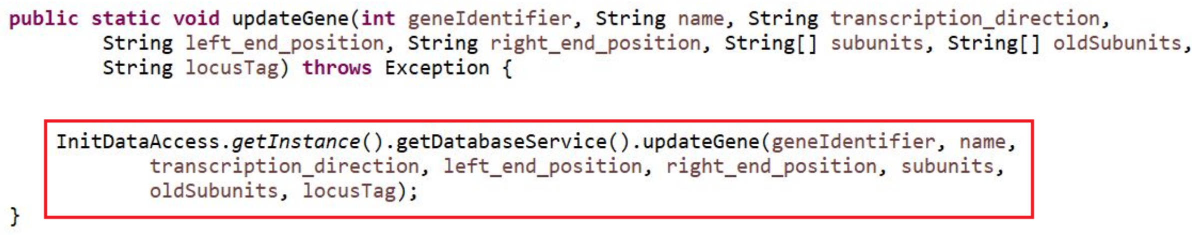
It is in these services that all the business logic is made, making it not dispersed by the application. Thus, any change in business logic is made only in this subpackage.

4.2.6 *Business logic layer integration*

After the services implementation, it was necessary to integrate them. Thus, the database access API classes were migrated to the ”merlin-services” project, and all the logic involved was replaced by the services.

In other words, the ”merlin-services” project bridges between previously created services and GUI calls. The way services are called in ”merlin-services” project is shown in figure 22.

Once the APIs were replaced by the services, it was necessary to perform a major refactor of all the projects dependent on APIs, to use the created services, becoming dependent on



```
public static void updateGene(int geneIdentifier, String name, String transcription_direction,
        String left_end_position, String right_end_position, String[] subunits, String[] oldSubunits,
        String locusTag) throws Exception {

    InitDataAccess.getInstance().getDatabaseService().updateGene(geneIdentifier, name,
            transcription_direction, left_end_position, right_end_position, subunits,
            oldSubunits, locusTag);
}
```

Figure 22.: Example of services integration.

the "merlin-services" projects. The refactoring alone was such an endeavor that it was out of the scope of this project, being implemented by a task force.

4.3 merlin architecture: pos-implementation

The changes mentioned above, it was possible to obtain an application with a new architecture, structured by layers, that allows to remove the previously existing coupling between database, business logic and graphical interface layers.

The comparison of the previous merlin database connection architecture with the implemented database connection architecture is shown in figures 23 and 24.

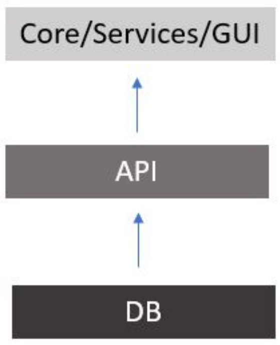


Figure 23.: Previous *merlin* database connection architecture.

As shown in figure 23, the business layer (API) interacts directly with the database. These interactions are performed through SQL queries. It can also be seen that the application core, services, and GUI are all in the same layer, which can cause several issues as the components are coupled and one change implies changes across all components.

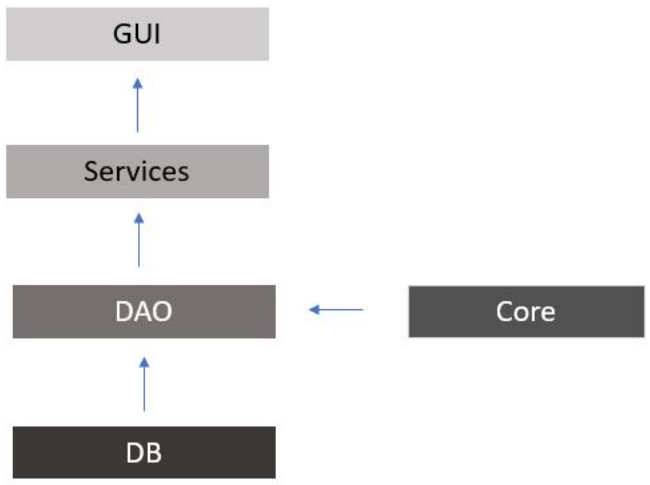


Figure 24.: Improved *merlin* database connection architecture.

On the other hand, in figure 24 all layers are separated. The interaction with the database is performed by the DAOs layer. At a higher level it has the services layer, where all the business logic is implemented. This layer is now separated from the core and GUI. This layers separation brings great advantages that will be approached later.

4.3.1 *Application Workspace Architecture*

The new project workspace architecture is shown in figure 25. Almost all the projects were renamed and new projects were created, as can be seen by comparing with previous architecture shown in figure 9.

The code related to the GUI is available in the ″aibench″ project, whereas, all processes necessary to GUI communicate with services are provided in the ″processes″ project.

All the logic relative to sequence alignments is in ″alignments″ project.

All application business logic is in the ″services″ project, while external services are in the ″bioapis″ project. The ″utilities″ project contains utilities required by the application and does not have any dependencies.

The project ″hibernate″ contains all the code required for the connection between the services and the database layers.

The ″core″ project contains the containers, datatypes, interfaces and utilities.

External containers are in the ”biocomponents” project, which has only external dependencies.

All projects on which the application is not directly dependent have been separated and are identified as plug-ins, such as ”gpr”, ”interpro”, ”biomass” and ”compartments” projects.

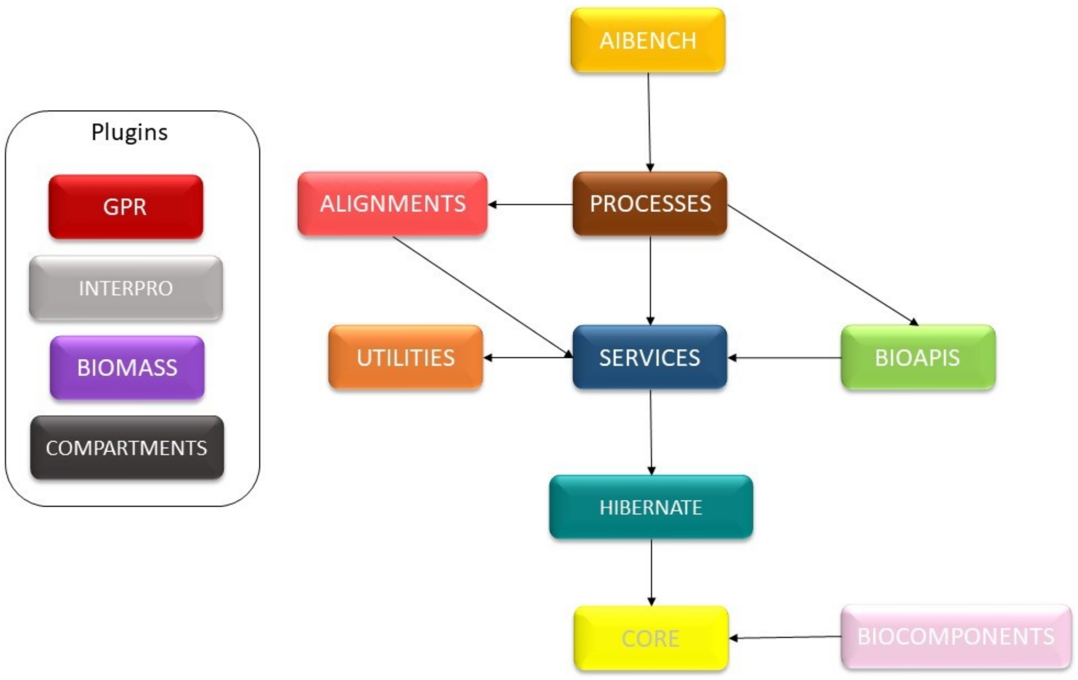


Figure 25.: Improved project architecture.

The comparison of the previous and the new architecture, shows that all the GUI logic was in the ”core” project, which was at the top level of the architecture. Whereas now the GUI logic has been separated from the ”core” project, and is now in the ”aibench” project. ”aibench” is currently the top level project of the entire project architecture, while the ”core” project is now at the lowest level.

These changes allowed to organize the software architecture in layers, resulting in better code organization, allowing teamwork to be made easier.

4.3.2 *APIs*

As mentioned before, when the services integration, the APIs were replaced by the services present in the "merlin-services" project.

All business logic previously present in the APIs has been migrated to this project, with no changes in the application business logic, only in the way it is organized and how the information is returned by the database.

These services do not use the SQL queries previously presented in APIs, but use DAOs created using the Criteria API. Therefore, the code is now more organized and more easily reusable as services use these DAOs, which have dynamic parameters.

Thus, in order to replace the SQL queries present in "CompartmentsAPI", 3 DAOs were created and about 40 database access methods were developed. These methods include operations such as:

- insert compartments if missing;
- update compartments and abbreviations;
- retrieve all locus tags of a given gene;
- retrieve all enzymatic reactions that occur in a given compartment;
- delete reactions;
- get the number of reactants in a given compartment;
- get all the compartments;
- get compartment data for a given name;
- calculate total number of compartments;
- retrieve all information from compartment table;
- get reaction IDs and EC numbers;
- get reactions by source;
- check the existence of biochemical reactions;
- check the existence of transporters reactions;
- remove all reactions assigned to the model;
- remove biochemical reactions assigned to the model;

To replace the SQL queries present in "HomologyAPI", 23 DAOs were created and about 260 database access methods were developed. These methods include operations such as:

- retrieve homology availabilities for gene key;
- retrieve InterPro availability for gene key;
- get homology results for gene key;
- retrieve homologies taxonomy for gene key;
- retrieve InterPro results for key;
- retrieve loaded InterPro annotations with a given status;
- load InterPro location features;
- delete a set of genes from homology searches;
- retrieve genes available in homology database table;
- calculate total number of genes;
- count number of homologues genes;
- get taxonomy data;
- get locus tag and UniProt EC number
- delete the configurations of a specific database table
- retrieve the maximum taxonomic rank for an organism;
- get all gene information;
- get blast information;
- get enzymes by reaction;
- get enzymes by given reaction and pathway;
- get protein ID and EC number for a given gene ID;
- get protein ID and EC number for a given reaction.

To replace the SQL queries present in "ModelAPI", 44 DAOs were created and about 550 database access methods were developed. These methods include operations such as:

- retrieve all gene name aliases;

- retrieve all EC numbers associated to each gene locus tag.
- retrieve all products;
- retrieve all products aliases;
- retrieve all pathways and the enzymes associated to each pathway;
- load enzyme information;
- retrieve the compartments database identifiers;
- determine if compartment information for gene is loaded;
- retrieve reaction containers associated to reactions;
- return existing compartments;
- retrieve the compartments allocation for a given enzyme;
- update EC number status;
- get locus tag orthologs from database;
- run gene-protein reactions assignment;
- get information for e-biomass;
- add the biomass pathway to model;
- remove the selected reaction;
- get active reactions;
- get proteins data;
- get reactions data;
- get pathways;
- get compounds that participate in reactions;
- get compounds reversibility;
- calculate which metabolites have both properties (product and reactant);
- count distinct KEGG reactions;
- count distinct reactions in model inserted by homology;

To replace the SQL queries present in ”ProjectAPI”, about 140 database access methods were developed in the previous created DAOs, as this API contained all methods that could be related to more than one API. These methods include operations such as:

- check if is the transported search performed;
- check if are the transporters loaded;
- check if transporters are integrated in internal database;
- get the name of a reaction for a given ID;
- get compounds that participate in reactions;
- gets compounds with biological roles;
- get pathway ID, code and name;
- count reactions by pathway ID;
- count compounds with a given name;
- get data of a given metabolite;
- get reactions related with a given metabolite;
- check if the the information already exists in the table;
- check if a reaction of a given name exists;
- check if any gene exists;
- count the number of genes that encode proteins;
- count the number of genes that encode enzymes and transporters;
- calculate number of proteins associated to genes;
- retrieve the expected value used during BLAST;

Once these APIs were replaced by the created services, they are no longer needed so they have been deleted.

4.3.3 *Internal databases schema*

Currenty, merlin internal database uses a different schema from the initial one. The schema initially used by merlin can be viewed at https://gitlab.bio.di.uminho.pt/merlin-sysbio/merlin-core/tree/master/utilities.

However, the previous relational model limited the implementation of new features. Hence, throughout this thesis, the following changes were made to the database to improve database performance, organization and understanding:

- All key indexing and entities relationships have been carefully analyzed and, when necessary, corrected to ensure that they were correctly implemented.
- Changes were made to some queries to improve performance.
- All views have been removed, as tables containing the data have been optimized to search them directly.
- "chromossome" entity and all transporter-related entities were removed, since merlin did not currently use these entities.

  These changes translate to better database performance.

5 CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

Throughout this thesis, the object-relational mapping tool Hibernate was implemented in merlin's framework. This practical application had as goal to solve the problems present in the application at dependencies' level, removing MySQL^{TM}and H2 databases dependencies from merlin's core.

To make this implementation possible, it was necessary to create, in a first phase, database access objects using the JPA Criteria API, making the query language adaptable to any type of database. Finally, the application business logic have been reimplemented in services, which use the created database access objects, making the database type abstract to the application.

The mentioned improvements were successfully developed and integrated into merlin's framework.

These implementations contributed to the objective of making the application independent from the database. Also, they provided a simple and easily accessible architecture, making the application code easier to understand and to maintain.

The application has become scalable in terms of database, adaptable to any type, just by changing the configuration of the connection to the database. Hibernate manages the database connections and provides a caching mechanism, making the database access faster.

This implementation involved restructuring the architecture of the application, making it truly modular. Hence, a change in one layer does not necessarily imply changes in all other layers.

In addition, the fact that the methods use the Criteria API will greatly facilitate development.

Although this implementation has been successful and has brought various advantages to merlin, there are still certain weaknesses.

Hibernate is a third-party library, which is susceptible to bugs and deprecation.

Also, some operations still lack a data model between the GUI and the data layer. Integers, strings, arrays should be replaced by objects, which ease the classification of the type of information retrieved from the data layer.

The developed code was integrated in merlin current version, and the application was used with this implementation in the course ”S2M2 - Summer School in Metabolic Modeling”, held at the University of Minho in June 2019. Currently, it continues being developed in the Bioinformatics and Systems Biology lab of the BioSystems groups at University of Minho.

All developed code is available in the merlin 4.0 repository and can be accessed through the link https://gitlab.bio.di.uminho.pt/merlin4/merlin-hibernate/tree/dev_spereira.

Finally, these changes improved merlin's usability.

5.2 FUTURE WORK

There are still many improvements that can be made to the application.

It would be important to rethink the way data is returned from the APIs. Currently, in some operations it is returned as simple data, that is, strings, arrays, integers, but it would be more useful for the application to be returned as objects, facilitating the access to the information.

Another possible improvement would be in the use case where the merlin application access to the database remotely. It would be a great improvement, having a version with access to a remote server that has the business logic layer and the entire data layer implemented on the server side and not in the application. This way, it would avoid cases of updates to the application due to bugs in the two layers mentioned above and, in addition, a remote server has a much higher performance, being important since merlin is becoming an application more and more used by several people at a time. It would be possible with the development and integration of Spring services, once it allows to create autoinjection for the Restful controllers layer. Therefore, with Spring services integration, it would allow to create a merlin version, holding a Restful API, where some of the plug-ins may be on a server performing heavier computational tasks, as well as creating a graphical interface based on web interfaces.

BIBLIOGRAPHY

[1] Augusto, T. & Barbosa, P. Biologia Sistêmica e Ensino de Ciências: Um novo paradigma ou neo-positivismo? *XI Encontro Nacional de Pesquisa em Educação em Ciências Universidade Federal de Santa Catarina, Florianópolis, SC* 1–8 (Julho 2017).

[2] Mesquita, E. & et al. The Paradigm of Systems Biology Applied to Cardiovascular Diseases. *International Journal of Cardiovascular Sciences* **1(1)**, 78–86 (2015).

[3] de Araújo, N. & et al. A Era Da Bioinformática : Seu Potencial E Suas Implicações Para As Ciências Da Saúde. *Estudos de Biologia* **30(70/72)**, 143–148 (January 2008).

[4] Dias, O., Rocha, M., Ferreira, E. & Rocha, I. Merlin: Metabolic models reconstruction using genome-scale information. *Proceedings of the 11 International Symposium on Computer Applications in Biotechnology* **IFAC Proceedings Volumes 43(6)**, 120–125 (2010).

[5] Agren, R. & et al. The RAVEN Toolbox and Its Use for Generating a Genome-scale Metabolic Model for Penicillium chrysogenum. *PLoS Computational Biology* **9(3)**, e1002980 (March 2013).

[6] Orth, J. *et al.* A comprehensive genome-scale reconstruction of Escherichia coli metabolism-2011. *Molecular Systems Biology* **7**, 1–9 (October 2011).

[7] merlin. merlin: metabolic models reconstruction using genome-scale information. Available in: `https://merlin-sysbio.org/index.php/Home`. Accessed January 10, 2019.

[8] Dias, O., Rocha, M., Ferreira, E. & Rocha, I. Reconstructing genome-scale metabolic models with merlin. *Nucleic Acids Research* **43(8)**, 3899–3910 (April 2015).

[9] Red Hat, I. Hibernate Everything data. Available in: `http://hibernate.org/`. Accessed January 10, 2019.

[10] Potters, G. Systems Biology of the Cell. *Nature Education 3(9):33* (2010).

[11] Alon, U. *An Introdution To Systems Biology: design principles of biological circuits* (CRC Press, 2006), 1 edn.

[12] Kitano, H. Systems Biology: A Brief Overview. *Science* **295**, 1662–1664 (March 2002).

[13] Kitano, H. Computational systems biology. *Nature International Journal of Science* **420**, 206–210 (November 2002).

[14] Fondi, M. & Liò, P. Genome-Scale Metabolic Network Reconstruction. *Methods in Molecular Biology* **1231**, 233–256 (January 2015).

[15] Bordbar, A., Monk, J., King, Z. & Palsson, B. Constraint-based models predict metabolic and associated cellular functions. *Nature Reviews Genetics* **15(2)**, 107–120 (February 2014).

[16] Kim, T., Sohn, S., Kim, Y., Kim, W. & Lee, S. Recent advances in reconstruction and applications of genome-scale metabolic models. *Current Opinion in Biotechnology* **23(4)**, 617–623 (August 2012).

[17] Feist, A. M., Herrgrd, M. J., Thiele, I., Reed, J. L. & Palsson, B. Ø. Reconstruction of biochemical networks in microorganisms. *Nature Reviews Microbiology* **7(2)**, 129–143 (December 2008).

[18] Xu, C. *et al.* Genome-scale metabolic model in guiding metabolic engineering of microbial improvement. *Applied Microbiology and Biotechnology* **97(2)**, 519–539 (January 2013).

[19] Rocha, I., Förster, J. & Nielsen, J. Design and Application of Genome-Scale Reconstructed Metabolic Models. *Methods in molecular biology* **416**, 409–431 (2008).

[20] Baart, G. & Martens, D. Genome-Scale Metabolic Models: Reconstruction and analysis. *Methods in Molecular Biology* **799**, 107–126 (2012).

[21] Baart, G. & Martens, D. Genome-Scale Metabolic Models : Reconstruction and Analysis. *Neisseria meningitidis. Methods in Molecular Biology (Methods and Protocols)* **799**, 107–126.

[22] Kanehisa, M. & Goto, S. KEGG: kyoto encyclopedia of genes and genomes. *Nucleic acids research* **28(1)**, 27–30 (January 2000).

[23] International, S. Biocyc database collection. Available in: `https://biocyc.org/`. Accessed November 13, 2018.

[24] Loira, N., Zhukova, A. & Sherman, D. Pantograph: A template-based method for genome-scale metabolic model reconstruction. *Journal of Bioinformatics and Computational Biology* **13(2)**, 1550006 (April 2015).

[25] Hucka, M. & et al. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics* **19(4)**, 524–531 (March 2013).

[26] Dubitzky, W. and Wolkenhauer, O. and Yokota, H. and Cho, K. *Encyclopedia of Systems Biology* (Springer-Verlag, 2013).

[27] Prlic, A. & et al. BioJava: an open-source framework for bioinformatics in 2012. *Bioinformatics* **28(20)**, 2693–2695 (October 2012).

[28] Patient, S. *et al.* UniProtJAPI: a remote API for accessing UniProt data. *Bioinformatics* **24(10)**, 1321–1322 (May 2018).

[29] H2 database. Available in: https://http://www.h2database.com/html/main.html. Accessed November 07, 2018.

[30] Mysql 5.7 reference manual. Available in: https://dev.mysql.com/doc/refman/5.7/en/what-is-mysql.html. Accessed November 07, 2018.

[31] Dias, O., Rocha, M., Ferreira, E. & Rocha, I. Reconstructing High-Quality Large-Scale Metabolic Models with merlin. *Methods in Molecular Biology* **1716**, 1–36 (January 2018).

[32] H2 database - introduction. Available in: https://www.tutorialspoint.com/h2_database. Accessed November 07, 2018.

[33] Almeida, F. *Practical SQL Guide for Relational Databases* (2016).

[34] C., B. & King, G. *Hibernate in Action* (Manning Publications, 2004).

[35] da Silva, C. & et al. Estratégias de Persistência em Software Orientado a Objetos: Definição e Implementação de um Framework para Mapeamento Objeto-Relacional. Bachelor in Information Systems - Faculdade Metodista Granbery, Juiz de Fora .

[36] Inc., I. T. ObjectStore® Standard Edition: The Database Behind the Worlds Most Scalable Applications. Available in: https://www.ignitetech.com/solutions/information-technology/objectstore. Accessed January 13, 2019.

[37] Corporation, A. Actian NoSQL Object Database The New Versant. Available in: https://www.actian.com/data-management/nosql-object-database/e. Accessed January 13, 2019.

[38] Inc., O. Objectivity/DB. Available in: https://www.objectivity.com/products/objectivitydb/. Accessed January 13, 2019.

[39] Srinivasan, V. & Chang, D. Object persistence in object-oriented applications. *IBM Systems Journal* **36(1)**, 66–87 (1997).

[40] Sardagna, M. & Vahldick, A. Aplicação do Padrão Data Access Object (DAO) em Projetos Desenvolvidos com Delphi. Graduation work - Universidade Regional de Blumenau (FURB) Blumenau, SC, Brasil .

[41] Oracle. Core j2ee patterns - data access object. Available in: https://www.oracle.com/technetwork/java/dataaccessobject-138824.html. Accessed October 23, 2018.

[42] Best Practice Software Engineering: Data Access Object. Available in: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/dao.html. Accessed October 10, 2018.

[43] Patni, S. *Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS* (California: Apress®, March 2017), 1 edn.

[44] Junjin, M. An Approach for SQL Injection Vulnerability Detection. *Sixth International Conference on Information Technology: New Generations* 1411–1414 (April 2009).

[45] Thomas, S. & Williams, L. Using Automated Fix Generation to Secure SQL Statements. *SESS '07 Proceedings of the Third International Workshop on Software Engineering for Secure Systems* 9 (May 2007).

[46] Thomas, S., Williams, L. & Xie, T. On automated prepared statement generation to remove SQL injection vulnerabilities. *Information and Software Technology* **51(3)**, 589–598 (March 2009).

[47] Derr, R. Vulnerabilities. *Threat Assessment and Risk Analysis* 83–95 (2016).

[48] Oracle. Oracle - Java documentation. Available in: https://docs.oracle.com/javase/tutorial/jdbc/basics/processingsqlstatements.html. Accessed October 23, 2018.

[49] Group, T. P. PHP documentation: Prepared Statements. Available in: http://php.net/manual/en/mysqli.quickstart.prepared-statements.php. Accessed October 26, 2018.

[50] Corporation, O. 13.5 Prepared SQL Statement Syntax. Available in: https://dev.mysql.com/doc/refman/8.0/en/sql-syntax-prepared-statements.html. Accessed November 1, 2018.

[51] Chen, T. & et al. Detecting performance anti-patterns for applications developed using object-relational mapping. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014* 1001–1012 (May 2014).

[52] ORM : Object Relational Mapper. Available in: https://www.devmedia.com.br/orm-object-relational-mapper/19056. Accessed October 23, 2018.

[53] Faria, T. *Java EE 7 com JSF, PrimeFaces e CDI*, vol. 1 (Algaworks Editora, 2013), 2 edn.

[54] MyBatis.org. Mybatis 3: Introduction. Available in: http://www.mybatis.org/mybatis-3/. Accessed February 05, 2019.

[55] Java2Blog. Introduction to hibernate framework. Available in: https://java2blog.com/introduction-to-hibernate-framework/. Accessed November 10, 2018.

[56] da Silva, J. and et al. Persisência de dados em java utilizando Hibernate. *Revista Interface Tecnológica* **6(1)**, 33–43 (2009).

[57] Richardson, C. ORM in Dynamic Languages. *ACM Queue* **6(3)**, 28–37 (June 2008).

[58] Miiller, M. & Bonetti, T. Mapeamento Objeto Relacional Com Hibernate Em Aplicações Java Web (2005).

[59] Inc., R. H. Hibernate EntityManager. Available in: https://docs.jboss.org/hibernate/entitymanager/3.6/reference/en/html_single/. Accessed January 14, 2019.

[60] 5 new features in hibernate 5 every developer should know. Available in: https://www.thoughts-on-java.org/5-new-features-hibernate-5-every-developer-know/. Accessed November 7, 2018.

[61] Phutela, D. Hibernate Vs JDBC. *Mindfire Solutions* 1–8 (2012).

[62] Corporation, O. Mysql: The world's most popular open source database. Available in: https://www.mysql.com/. Accessed January 30, 2019.

[63] Microsoft. Sql server: Execute o sql server na sua plataforma favorita. Available in: https://www.microsoft.com/pt-pt/sql-server/sql-server-2017. Accessed January 30, 2019.

[64] Group, T. P. G. D. Postgresql: The world's most advanced open source relational database. Available in: https://www.postgresql.org/. Accessed January 30, 2019.

[65] Oracle. Oracle database. Available in: https://www.oracle.com/database/. Accessed January 30, 2019.

[66] Hibernate - criteria queries. Available in: https://www.tutorialspoint.com/hibernate/hibernate_criteria_queries.htm. Accessed November 12, 2018.

[67] Hibernate orm 5.2.16.final user guide. Available in: https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html. Accessed November 7, 2018.

[68] Oracle. The Java EE 6 Tutorial. Available in: https://docs.oracle.com/javaee/6/tutorial/doc/gjrij.html. Accessed October 27, 2018.

[69] Inc., Q. Using criteria in hibernate for advanced queries. Available in: https://www.developer.com/db/using-criteria-in-hibernate-for-advanced-queries.html. Accessed November 8, 2018.

[70] Bhatti, S., Abro, Z. & Abro, F. Performance evaluation of java based object relational mapping tool. *Mehran University Research Journal of Engineering and Technology* **32(2)**, 159–166 (January 2013).

[71] Red Hat, I. Hibernate Chapter 9. Criteria Queries. Available in: https://docs.jboss.org/hibernate/orm/4.0/hem/en-US/html/querycriteria.html. Accessed October 27, 2018.

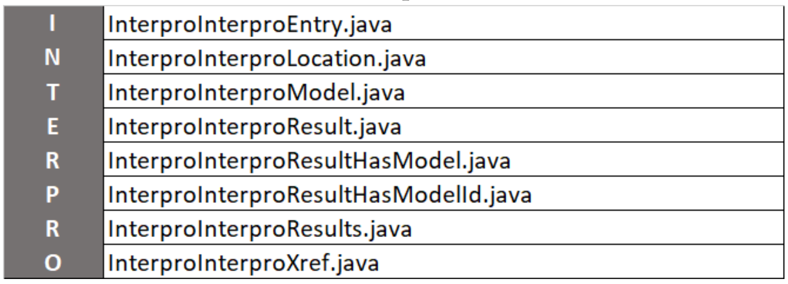
[72] Glez-Penã, D. *et al.* AIBench: A rapid application development framework for translational research in biomedicine. *Computer methods and programs in biomedicine* **8**, 191–203 (December 2009).

Appendices

A

MERLIN INTERPRO ENTITIES

Table 1.: Interpro Entities.

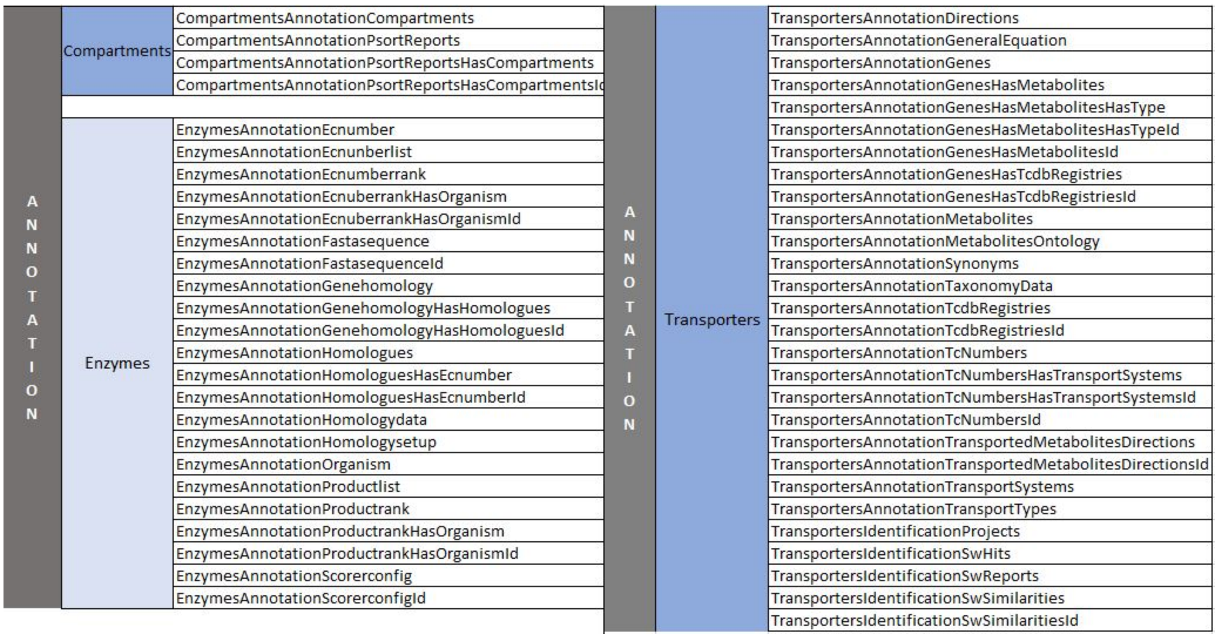


I	InterproInterproEntry.java
N	InterproInterproLocation.java
T	InterproInterproModel.java
E	InterproInterproResult.java
R	InterproInterproResultHasModel.java
P	InterproInterproResultHasModelId.java
R	InterproInterproResults.java
O	InterproInterproXref.java

B

MERLIN ANNOTATION ENTITIES

Table 2.: Annotation Entities.



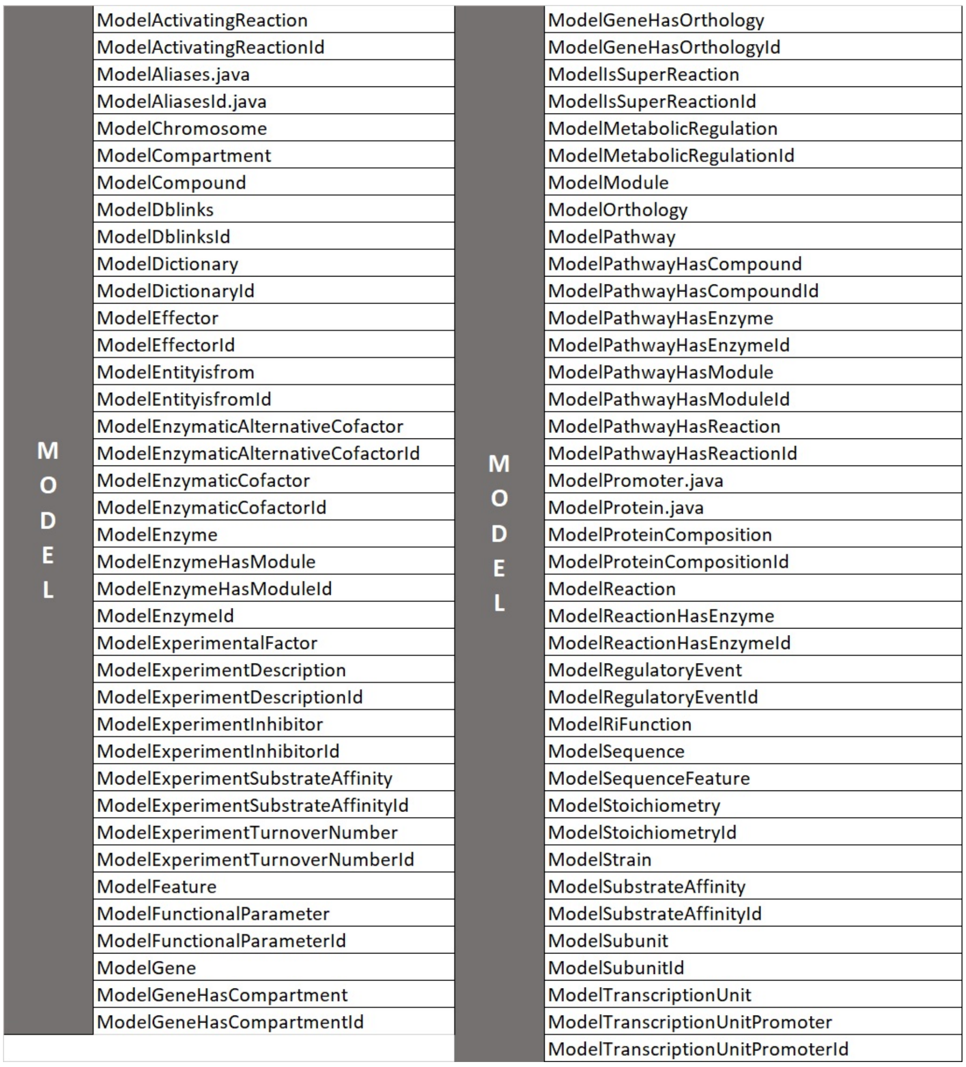
ANNOTATION	Compartments	CompartmentsAnnotationCompartments
		CompartmentsAnnotationPsortReports
		CompartmentsAnnotationPsortReportsHasCompartments
		CompartmentsAnnotationPsortReportsHasCompartmentsI
	Enzymes	EnzymesAnnotationEcnumber
		EnzymesAnnotationEcnunberlist
		EnzymesAnnotationEcnumberrank
		EnzymesAnnotationEcnuberrankHasOrganism
		EnzymesAnnotationEcnuberrankHasOrganismId
		EnzymesAnnotationFastasequence
		EnzymesAnnotationFastasequenceId
		EnzymesAnnotationGenehomology
		EnzymesAnnotationGenehomologyHasHomologues
		EnzymesAnnotationGenehomologyHasHomologuesId
		EnzymesAnnotationHomologues
		EnzymesAnnotationHomologuesHasEcnumber
		EnzymesAnnotationHomologuesHasEcnumberId
		EnzymesAnnotationHomologydata
		EnzymesAnnotationHomologysetup
		EnzymesAnnotationOrganism
		EnzymesAnnotationProductlist
		EnzymesAnnotationProductrank
		EnzymesAnnotationProductrankHasOrganism
		EnzymesAnnotationProductrankHasOrganismId
		EnzymesAnnotationScorerconfig
		EnzymesAnnotationScorerconfigId

ANNOTATION	Transporters	TransportersAnnotationDirections
		TransportersAnnotationGeneralEquation
		TransportersAnnotationGenes
		TransportersAnnotationGenesHasMetabolites
		TransportersAnnotationGenesHasMetabolitesHasType
		TransportersAnnotationGenesHasMetabolitesHasTypeId
		TransportersAnnotationGenesHasMetabolitesId
		TransportersAnnotationGenesHasTcdbRegistries
		TransportersAnnotationGenesHasTcdbRegistriesId
		TransportersAnnotationMetabolites
		TransportersAnnotationMetabolitesOntology
		TransportersAnnotationSynonyms
		TransportersAnnotationTaxonomyData
		TransportersAnnotationTcdbRegistries
		TransportersAnnotationTcdbRegistriesId
		TransportersAnnotationTcNumbers
		TransportersAnnotationTcNumbersHasTransportSystems
		TransportersAnnotationTcNumbersHasTransportSystemsId
		TransportersAnnotationTcNumbersId
		TransportersAnnotationTransportedMetabolitesDirections
		TransportersAnnotationTransportedMetabolitesDirectionsId
		TransportersAnnotationTransportSystems
		TransportersAnnotationTransportTypes
		TransportersIdentificationProjects
		TransportersIdentificationSwHits
		TransportersIdentificationSwReports
		TransportersIdentificationSwSimilarities
		TransportersIdentificationSwSimilaritiesId

C

MERLIN MODEL ENTITIES

Table 3.: Model Entities.



MODEL	ModelActivatingReaction	MODEL	ModelGeneHasOrthology
	ModelActivatingReactionId		ModelGeneHasOrthologyId
	ModelAliases.java		ModelIsSuperReaction
	ModelAliasesId.java		ModelIsSuperReactionId
	ModelChromosome		ModelMetabolicRegulation
	ModelCompartment		ModelMetabolicRegulationId
	ModelCompound		ModelModule
	ModelDblinks		ModelOrthology
	ModelDblinksId		ModelPathway
	ModelDictionary		ModelPathwayHasCompound
	ModelDictionaryId		ModelPathwayHasCompoundId
	ModelEffector		ModelPathwayHasEnzyme
	ModelEffectorId		ModelPathwayHasEnzymeId
	ModelEntityisfrom		ModelPathwayHasModule
	ModelEntityisfromId		ModelPathwayHasModuleId
	ModelEnzymaticAlternativeCofactor		ModelPathwayHasReaction
	ModelEnzymaticAlternativeCofactorId		ModelPathwayHasReactionId
	ModelEnzymaticCofactor		ModelPromoter.java
	ModelEnzymaticCofactorId		ModelProtein.java
	ModelEnzyme		ModelProteinComposition
	ModelEnzymeHasModule		ModelProteinCompositionId
	ModelEnzymeHasModuleId		ModelReaction
	ModelEnzymeId		ModelReactionHasEnzyme
	ModelExperimentalFactor		ModelReactionHasEnzymeId
	ModelExperimentDescription		ModelRegulatoryEvent
	ModelExperimentDescriptionId		ModelRegulatoryEventId
	ModelExperimentInhibitor		ModelRiFunction
	ModelExperimentInhibitorId		ModelSequence
	ModelExperimentSubstrateAffinity		ModelSequenceFeature
	ModelExperimentSubstrateAffinityId		ModelStoichiometry
	ModelExperimentTurnoverNumber		ModelStoichiometryId
	ModelExperimentTurnoverNumberId		ModelStrain
	ModelFeature		ModelSubstrateAffinity
	ModelFunctionalParameter		ModelSubstrateAffinityId
	ModelFunctionalParameterId		ModelSubunit
	ModelGene		ModelSubunitId
	ModelGeneHasCompartment		ModelTranscriptionUnit
	ModelGeneHasCompartmentId		ModelTranscriptionUnitPromoter
			ModelTranscriptionUnitPromoterId