

**FACULTY
OF MATHEMATICS
AND PHYSICS**
Charles University

BACHELOR THESIS

Lukáš Polák

**Extension of web-based interface for
protein binding sites prediction**

Department of Software Engineering

Supervisor of the bachelor thesis: doc. RNDr. David Hoksza, Ph.D.

Study programme: Programming and software
development Bc.

Study branch: IPP2

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In date

Author's signature

I would like to thank doc. RNDr. David Hoksza, Ph.D. for supervising my thesis, and introducing me to bioinformatics. I appreciate the useful consultations and feedback I received. Thanks to Mgr. Petr Škoda, Ph.D. for consultations related to PrankWeb architecture. I also want to thank my family for all of their support during my studies.

Title: Extension of web-based interface for protein binding sites prediction

Author: Lukáš Polák

Department: Department of Software Engineering

Supervisor: doc. RNDr. David Hoksza, Ph.D., Department of Software Engineering

Abstract: Protein-ligand binding sites are positions on the protein structure where the protein interacts with other molecules. PrankWeb is a web server developed at MFF UK allowing prediction of such places. These predictions are essential in fields such as bioengineering and computational drug discovery. The goal of this thesis was to update this web server, i.e., replace old and unsupported components with new ones. Another goal was to extend the server architecture to enable the simple addition of modules for the postprocessing of the predicted binding sites. These modules can be implemented either on the client side in case of simple computations, or on the server side in case of complex computations. As part of the thesis, we have implemented a client module for computing the volume of active sites and a server module allowing the docking of small proteins into predicted binding sites. The thesis describes not only the interventions in the architecture but also provides a short introduction to the problem of protein-ligand binding sites and their prediction.

Keywords: bioinformatics web software protein

Contents

Introduction	3
1 Introduction and background	5
1.1 Introduction to molecular biology	5
1.1.1 Amino acids and proteins	5
1.1.2 Protein functions	6
1.2 Used data formats	7
1.2.1 JSON	7
1.2.2 PDB	7
1.2.3 PDBx/mmCIF	9
1.2.4 FASTA	9
1.2.5 CSV	10
1.3 P2Rank tool	10
1.4 PrankWeb architecture	10
1.4.1 Gateway	13
1.4.2 RabbitMQ	13
1.4.3 Flower	13
1.4.4 Web-server	14
1.4.5 Executor-P2Rank	14
1.4.6 Executor-Docking	15
1.4.7 Prometheus	15
1.5 Similar web-tools	15
1.5.1 IntFOLD	15
1.5.2 COACH	16
1.5.3 DeepSite	17
2 Programming documentation	19
2.1 Frontend	19
2.1.1 High-level overview	20
2.1.2 MolStar	21
2.1.3 RCSB Saguaro 1D Feature Viewer	27

2.1.4	React components	30
2.2	Plug-ins	31
2.2.1	Client-side plug-ins	32
2.2.2	Server-side plug-ins	34
3	User documentation	41
3.1	Deployment	41
3.1.1	Docker deployment	41
3.1.2	Local deployment	43
3.2	User guide	44
3.3	Developer	47
3.3.1	Client-side plug-ins	48
3.3.2	Server-side plug-ins	49
	Conclusion	51
	Bibliography	53
A	Attachments	57
A.1	Source codes	57
A.2	GitHub	57
A.3	Abbreviations	58
A.4	Pocket detail designs	58

Introduction

Detection of protein ligand-binding sites is a vital aspect of nowadays drug discovery and development. Identification of the potential binding sites allows an understanding of various molecule interactions, which is the first step of rational drug discovery pipelines. Recognizing the binding sites is thus very important for further bioinformatic research and studies [1].

P2Rank is a machine-learning-based tool developed at MFF UK which allows users to predict the ligand-binding sites for a given protein structure. P2Rank works standalone and outperforms most of the existing binding sites prediction tools [2]. PrankWeb is a web-based tool that provides a user-friendly interface for P2Rank. A significant difference between PrankWeb and other web-based tools is that PrankWeb does not employ either JMol or JSMol for structure visualization of the results [3]. These libraries are rather old-fashioned. The original PrankWeb authors decided to use the LiteMol for structure visualization and Protael for sequence visualization. Although these libraries are still working, they are not actively developed anymore.

There were two main goals of this thesis. The first goal was to update the PrankWeb interface to use different libraries for the visualization, namely MolStar [4] and RCSB Saguaro 1D Feature Viewer [5]. This improves not only the visual appearance of the results but also the performance. Furthermore, the updated libraries are actively developed and thus potentially more reliable in the future. The second goal was to update the PrankWeb architecture, so that post-processing may be done on the predicted binding sites both on the client side and the server side. This introduces the potential to create custom plug-ins for the web interface.

The thesis presents a working version of the updated PrankWeb tool. The updated tool should keep the current functionality of the website and visualize the results via a more user-friendly interface. The tool should also be able to perform computations on the predicted binding sites.

In the first chapter, the reader will be introduced to the basics of protein and ligand-binding sites problematic, later in this chapter the current PrankWeb architecture and the P2Rank tool itself will be covered. The reader will also be briefly introduced to similar web tools. The second chapter will cover the

programming part of the work. Firstly, the usage of the updated libraries and other frontend design decisions will be introduced. Secondly, the plug-in architecture including both the client-side and the server-side computations will be covered in the respective subchapters. In the third chapter, the tool usage will be described from two perspectives - a user and a developer. The conclusion will cover the results and the potential extensions to the tool.

Chapter 1

Introduction and background

PrankWeb is a web-based tool for predicting ligand-binding sites for a given protein structure. PrankWeb uses a tool called P2Rank for the prediction of binding sites on the backend. In this thesis, an improved version of PrankWeb is introduced, with an emphasis on the modernization of the frontend and creating a potential for implementing post-processing features, such as docking.

In this chapter, we will discuss the basics of protein and ligand-binding sites problematic to get briefly acquainted with the topic. A short intro to the used data formats in this thesis will follow. Later in this chapter, we will cover the current PrankWeb architecture and the P2Rank tool itself to get a better understanding of the project. The reader will also be briefly introduced to similar web tools for the prediction of ligand-binding sites.

1.1 Introduction to molecular biology

In this section, we will briefly introduce the basics of molecular biology. We will focus on the protein structures and their interactions with other molecules. The reader is assumed to have a basic knowledge of protein structures and this terminology to understand the concepts of the entire thesis.

1.1.1 Amino acids and proteins

Amino acids are in general molecules containing an amino group ($-NH_2$), a carboxyl group ($-COOH$), and a R group (also called side chain). Although there are hundreds of known amino acids, 20 of them provide the key to the structure of thousands of different proteins. In this thesis, we will focus on the 20 proteinogenic amino acids that are used in the protein structures. Amino acids are the building blocks for proteins that serve as diverse products, such as enzymes, hormones,

antibodies, muscle fibers, transporters, and many more.

Proteins are polymers consisting of a linear chain of amino acids. Each of the so-called **amino-acid residues** is connected to its neighbor by a covalent bond. The connected amino acids form a **chain**. Proteins are occurring in all cells and all parts of the cells. Proteins are the molecular instruments through which genetic information is expressed. Cells may produce proteins with substantially different properties and activities just by joining them in a different **sequence** (also called **primary structure**). The 3D protein representation based on a sequence of the protein's amino acids is called the **tertiary structure** [6].

Billions of proteins are known by their sequence, but only a small fraction of them are known by their tertiary structure. The structures may be determined by experiments, we call these structures **experimental**. Despite the effort to determine the structures, the vast majority of actual protein structures remain unknown. Nowadays, various tools are used for predicting unknown structures. A notable example is a neural network called AlphaFold that allows the prediction of a protein structure based on its sequence by assigning a probability to each amino acid in the sequence (called pLDDT). We refer to these structures as **predicted** [7].

1.1.2 Protein functions

The functions of proteins depend not only on the protein structures but also on the environment in which they are located. Many protein functions are involved by the reversible binding to other molecules. A molecule that is reversibly bound to a protein is called a **ligand** (also **protein ligand**). A ligand may be any kind of molecule, including another protein, or a small molecule.

One ligand binds on a part of the protein surface. This is called the **ligand-binding site**. This binding is not only defined by its molecule, but also by other properties such as shape, size, and charge. We refer to the binding site as a **pocket** [6].

In PrankWeb, the main focus is to create a **prediction** of the potential locations and interacting residues of small molecule ligands for a given protein structure.

These predictions may or may not be based on various physical and chemical properties of spots on the protein surface, and **evolutionary conservation** score of the residues. The conservation score is a measure of how similar the residues are in different protein structures. In PrankWeb, evolutionary conservation is explicitly visualized.

One of the PrankWeb extensions is the possibility to **dock** a ligand to a binding site. Docking is a complex process of computing the mutual positions of the ligand and the protein structure. This computation also provides information about

the potential energy of the protein-ligand complex. This altogether allows us to predict the potential of real-life binding of the ligand to the protein structure [8].

1.2 Used data formats

PrankWeb uses several data formats for the input and output of data throughout the application. In this section, we will briefly introduce the used data formats.

1.2.1 JSON

JSON (JavaScript Object Notation) is a data format used for storing JavaScript objects. The syntax of this format is simple and the main advantages of this format are its readability and simple usage on the frontend. The main idea of JSON is to store its data in key-value pairs. The keys are strings and the actual values may be in any format such as string, number, boolean, array, or another object.

Listing 1.1 An example of a JSON file used for storing information about the prediction for the 2SRC protein structure.

```
1 {
2   "id": "2SRC",
3   "database": "v3",
4   "created": "2023-02-25T20:58:07",
5   "lastChange": "2023-02-25T20:58:26",
6   "status": "successful",
7   "metadata": {
8     "predictionName": "2SRC",
9     "structureName": "structure.cif"
10  }
11 }
```

1.2.2 PDB

PDB (Protein Data Bank format) is a file format that is used for describing three-dimensional protein structures. This format stores information about the atom coordinates and their connections [9]. Alongside this information, the PDB format may contain additional metadata as well. It is still used for many protein structures, but the format size is limited and thus it does not support large structures, so in some cases, it is replaced by the PDBx/mmCIF format (section 1.2.3) [10]. A large database of PDB files for protein structures is called RCSB PDB¹.

¹PDB database is available at <https://www.rcsb.org/>.

Listing 1.2 An edited example of a PDB file used for storing information about the 2SRC protein structure.

```

1 HEADER TYROSINE-PROTEIN KINASE 29-DEC-98 2SRC
2 TITLE CRYSTAL STRUCTURE OF HUMAN TYROSINE-PROTEIN KINASE C-SRC,
3 TITLE 2 IN COMPLEX WITH AMP-PNP
4 COMPND MOL_ID: 1;
5 COMPND 2 MOLECULE: TYROSINE-PROTEIN KINASE SRC;
6 COMPND 3 CHAIN: A;
7 COMPND 4 FRAGMENT: RESIDUES 86-836, CONTAINING SH2, SH3, KINASE 2
8 COMPND 5 DOMAINS AND C-TERMINAL TAIL;
9 COMPND 6 SYNONYM: C-SRC, P60-SRC;
10 COMPND 7 EC: 2.7.1.112;
11 COMPND 8 ENGINEERED: YES
12 ...
13 JRNL AUTH W.XU,A.DOSHI,M.LEI,M.J.ECK,S.C.HARRISON
14 JRNL TITL CRYSTAL STRUCTURES OF C-SRC REVEAL FEATURES OF ITS
15 JRNL TITL 2 AUTOINHIBITORY MECHANISM.
16 JRNL REF MOL.CELL V. 3 629 1999
17 JRNL REFN ISSN 1097-2765
18 JRNL PMID 10360179
19 JRNL DOI 10.1016/S1097-2765(00)80356-1
20 ...
21 DBREF 2SRC A 83 533 UNP P12931 SRC_HUMAN 85 535
22 SEQADV 2SRC PTR A 527 UNP P12931 TYR 529 MODIFIED RESIDUE
23 SEQRES 1 A 452 MET VAL THR THR PHE VAL ALA LEU TYR ASP TYR GLU SER
24 ...
25 MODRES 2SRC PTR A 527 TYR O-PHOSPHOTYROSINE
26 HET PTR A 527 16
27 HET ANP A 1 31
28 HETNAM PTR O-PHOSPHOTYROSINE
29 HETNAM ANP PHOSPHOAMINOPHOSPHONIC ACID-ADENYLATE ESTER
30 HETSYN PTR PHOSPHONOTYROSINE
31 FORMUL 1 PTR C9 H12 N 06 P
32 FORMUL 2 ANP C10 H17 N6 O12 P3
33 FORMUL 3 HOH *269(H2 O)
34 HELIX 1 1 SER A 134 TYR A 136 5 3
35 HELIX 2 2 ARG A 155 LEU A 162 1 8
36 HELIX 3 3 LEU A 223 TYR A 229 1 7
37 ...
38 SHEET 1 A 2 PHE A 86 ALA A 88 0
39 SHEET 2 A 2 VAL A 137 PRO A 139 -1 N ALA A 138 O VAL A 87
40 SHEET 1 B 3 THR A 129 PRO A 133 0
41 SHEET 2 B 3 TRP A 118 SER A 123 -1 N ALA A 121 O GLY A 130
42 SHEET 3 B 3 LEU A 108 ASN A 112 -1 N ASN A 112 O LEU A 120
43 ...
44 LINK N PTR A 527 C GLN A 526 1555 1555 1.33
45 LINK C PTR A 527 N GLN A 528 1555 1555 1.33
46 CISPEP 1 GLU A 332 PRO A 333 0 -0.29

```



```

47 SITE      1 AC1 20 LEU A 273 GLY A 276 VAL A 281 ALA A 293
48 SITE      2 AC1 20 LYS A 295 THR A 338 GLU A 339 TYR A 340
49 SITE      3 AC1 20 MET A 341 SER A 345 ASP A 386 ARG A 388
50 SITE      4 AC1 20 ASN A 391 LEU A 393 ASP A 404 HOH A1064
51 SITE      5 AC1 20 HOH A1147 HOH A1200 HOH A1208 HOH A1230
52 CRYST1    50.590  72.970 172.690 90.00 90.00 90.00 P 21 21 21 4
53 ...
54 ATOM      1 N   THR A 84      33.954 62.803 55.198 1.00 38.76      N
55 ATOM      2 CA  THR A 84      32.735 62.188 54.583 1.00 39.76      C
56 ATOM      3 C   THR A 84      32.941 60.717 54.186 1.00 37.91      C
57 ATOM      4 O   THR A 84      32.021 60.068 53.678 1.00 36.13      O
58 ATOM      5 CB  THR A 84      32.285 62.973 53.319 1.00 41.34      C
59 ATOM      6 OG1 THR A 84      33.424 63.240 52.489 1.00 41.84      O
60 ATOM      7 CG2 THR A 84      31.616 64.288 53.711 1.00 40.28      C
61 ATOM      8 N   THR A 85      34.140 60.192 54.419 1.00 35.88      N
62 ATOM      9 CA  THR A 85      34.437 58.806 54.075 1.00 34.00      C
63 ATOM     10 C   THR A 85      34.001 57.804 55.154 1.00 33.76      C
64 ...
65 CONECT 3547 3554
66 CONECT 3554 3547 3555
67 CONECT 3555 3554 3556 3558
68 ...
69 MASTER      276   0   2  17  15   0   5   6 3915   1  49  35
70 END

```

1.2.3 PDBx/mmCIF

PDBx/mmCIF (Protein Data Bank extended / macromolecular Crystallographic Information File) is an extension to the CIF format that stores crystallographic data. It was developed to overcome the limitations of the existing formats [11].

1.2.4 FASTA

FASTA (FAST-All) is a text format that describes either a nucleotide sequence or a protein sequence. In our case, this format is used in the second sense. This format is simple as it only contains a brief description of the protein on the first line starting with the ">" character and the sequence itself on the next line. The sequence is represented by a string of letters that represent the amino acids. This format was created for the FASTA program that was used for sequence alignment [12].

Listing 1.3 An example of a FASTA file used for storing information about the 7VNU sequence.

```
1 >7VNU_1|Chains A, B, C, D|Nucleoprotein|Severe acute respiratory syndrome
   coronavirus 2 (2697049)
2 GASNTASWFTALTQHGKEDLKFPRGQGVPIINTNSSPDDQIGYRRATRRIRGGDGKMKDLSRWY
   FYYLGTGPEAGLPYGANKDGIIVVATEGALNTPKDHIGTRNPANNAIIVLQLPQGTTLPGFYAE
```

1.2.5 CSV

CSV (Comma-Separated Values) is a text format that is used for storing tabular data in plain text. The first line of the file contains the column names and the following lines contain the actual data. The data are separated either by commas or semicolons. This format is simple and easy to use, but it does not support complex data types such as arrays or objects.

1.3 P2Rank tool

P2Rank allows its users to predict the ligand-binding sites for a given protein. In contrast to other projects, P2Rank was one of the first tools to employ machine learning for predicting pockets. Most of the other tools use geometry-based, energetic-based, or template-based methods. P2Rank outperforms most of the existing binding sites prediction tools [2]. Moreover, P2Rank works as a standalone application and is fully automated, which makes the tool very intuitive and easy to use.

P2Rank works with specific file formats such as PDB and PDBx/mmCIF. After running the tool on a specific protein structure file, the tool will provide a CSV output file with the prediction and residue-level scores. The output file includes predicted pockets, their ranks, center coordinates, adjacent residues, related surface atoms and a probability score.

The tool was written in Java and requires only the JRE² to run. Additionally, the source codes are publicly available at GitHub³. This allows the users to potentially modify the tool to their respective needs.

1.4 PrankWeb architecture

PrankWeb consists of several components that cooperate together. Currently, the application is deployed via Docker⁴ containers that are described in the Docker-

²Java Runtime Environment.

³Source codes for P2Rank are available at <https://github.com/rdk/p2rank>.

⁴Docker is a virtualization tool providing a stable interface for isolating and running applications. More information is available at <https://www.docker.com/>.

compose configuration file. The application consists of the following components:

- **gateway** - a reverse proxy that is responsible for routing the requests to the respective backend services and for serving the frontend
- **rabbitmq** - a broker that is used for communication between the web server and the backend services
- **flower** - a tool for monitoring the RabbitMQ broker and Celery workers
- **web-server** - a Flask server responsible for communication between the gateway and the RabbitMQ broker
- **executor-p2rank** - a backend service that is responsible for running the P2Rank tool, employs Celery workers
- **executor-docking** - a backend service that is responsible for running the docking tool, employs Celery workers
- **prometheus** - a tool for monitoring the Docker containers

A diagram of the architecture is shown in figure 1.1. This diagram is a simplified version of the C4 model [13] for PrankWeb, dark blue boxes represent the containers, light blue boxes represent the components, and the arrows represent the communication between the services. For the completed C4 model with more details, see the original wiki page⁵. The diagram includes retrieving information about the structures from external databases from AlphaFold [14] and RCSB PDB [15].

Some of the containers include environment variables that are required for the proper functionality. The environment variables are specified in the docker-compose configuration file. The user may modify these also by creating a `.env` file in the root directory of the project.

The docker-compose configuration file thus contains the entire PrankWeb logic. When employing a new plug-in or a new feature, a container may be introduced to this file to get easily integrated into the application.

Each of the containers is defined by a respective `Dockerfile`. Some containers are dependent on the order of deployment, some containers include a Docker volume definition that ensures the persistence of the data.

Now we will present the current Docker containers in more detail to get a broader knowledge of the architecture.

⁵The original wiki page is available at <https://github.com/cusbg/p2rank-framework/wiki/PrankWeb-architecture>

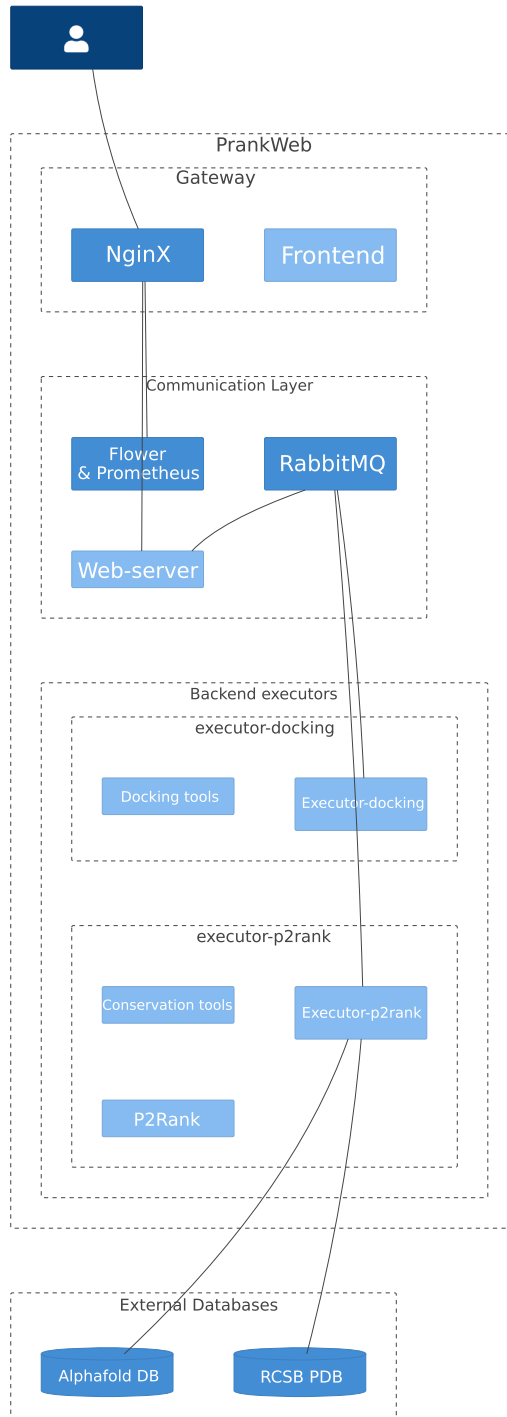


Figure 1.1 A simplified C4 model of the PrankWeb architecture.

1.4.1 Gateway

The gateway container is a reverse proxy that is responsible for routing the requests to the backend. In PrankWeb, we utilize Nginx⁶ as a reverse proxy. The Nginx configuration file is located in the `gateway/nginx.conf` file. The server configuration includes not only the reverse proxy routes but a mapping to the Flower and Prometheus services as well.

Moreover, `gateway/Dockerfile` is responsible for the installation of the frontend. The frontend is a React⁷ application built via webpack⁸. PrankWeb utilizes two main external libraries for the bioinformatic part of the application, MolStar and RCSB Saguaro Feature 1D Viewer. We will discuss these libraries in more detail in section 2.1.

The entire frontend is written in TypeScript, JavaScript, CSS, SCSS, and HTML.

1.4.2 RabbitMQ

RabbitMQ⁹ is a message broker that is used to provide communication between the web server and the backend workers, in our case Celery. RabbitMQ is configured via the respective configuration file. PrankWeb does not require a complex configuration of this service, although the service is still necessary for communication.

1.4.3 Flower

Flower¹⁰ is a tool for monitoring the broker and Celery¹¹ workers' functionality. The Flower container does not need any specific configuration, it is only necessary to correctly run it alongside the RabbitMQ container.

⁶Nginx is a web server used for serving static HTML content and acts as a reverse proxy. More information is available at <https://www.nginx.com/>.

⁷React is a library for creating modern and intuitive user interfaces for web browsers. More information is available at <https://react.dev/>.

⁸webpack is a module bundler used to bundle JavaScript for web browsers. More information is available at <https://webpack.js.org/>.

⁹More information about RabbitMQ is available at <https://www.rabbitmq.com/>.

¹⁰Source codes for Flower are available at <https://github.com/mher/flower>.

¹¹Celery is a distributed task queue for Python. This allows the users to distribute given tasks around multiple workers (in our case threads), so multiple tasks may run in parallel. Source codes are available at <https://github.com/celery/celery>.

1.4.4 Web-server

The web-server container is a WSGI server that is responsible for serving the web application. Currently, we employ the Gunicorn¹² server. The second main part of this container is Flask¹³ framework. The Flask application defines all of the REST API endpoints for interaction between the frontend and the backend. This application also defines a Celery client that is responsible for sending the tasks to the backend Celery workers based on the API calls.

1.4.5 Executor-P2Rank

This container is responsible for creating the prediction via the P2Rank tool. P2Rank executor is written in Python and utilizes the Celery framework for task management. Celery enables the server to use multiple threads and process the requests in parallel. The executor's Celery listener first receives a request for the prediction given a directory with the name of the structure. Subsequently, the executor prepares the necessary information for the P2Rank tool.

The tool is then run and the results are saved in the `predictions/<db-name>14/<structure-short>15/<structure-name> directory. The current hierarchy contains the following:`

- `input/configuration.json` - a JSON file required for a configuration of the P2Rank tool, containing the structure code, name of the structure file, conservation and others
- `public/structure.cif.gz` - a gzipped¹⁶ mmCIF/PDB file of the structure
- `public/prediction.json` - a JSON file containing the prediction results derived from the P2Rank tool specifically for easier parsing in the frontend
- `public/prankweb.zip` - a zip file containing unmodified, verbose output files directly from the P2Rank tool
- `info.json` - a JSON file containing current prediction status
- `log` - a log file containing the output of the P2Rank tool

¹²More information about Gunicorn is available at <https://gunicorn.org/>.

¹³Flask is a micro-framework for Python used for creating simple web applications. Source codes are available at <https://github.com/pallets/flask>.

¹⁴Represents the current version of database such as v2, v3, v3-alphaFold, v3-conservation-hmm etc.

¹⁵The shortcut consists of second two letters of the structure identifier, i.e SR for 2SRC or 5V for Q5VSL9.

¹⁶Gzip is an Unix-based tool for file compression.

All of the listed files are exposed via the REST API to the frontend. After posting a prediction request, the frontend periodically polls the `info.json` file to get the current status of the prediction. Meanwhile, the `log` file is continuously updated with the output of the P2Rank tool. The `log` file is also shown in the frontend to provide the user with the current status of the prediction.

1.4.6 Executor-Docking

This container is responsible for the docking backend task. More information about the docking task is available in section 2.2.2.

1.4.7 Prometheus

Prometheus¹⁷ is a tool for monitoring the Docker containers. It is not necessary for the proper application functionality but is useful for debugging purposes. The Prometheus container is configured via the respective configuration file. The interface is exposed at the 9090 port.

1.5 Similar web-tools

The main motivation for the creation of PrankWeb was to provide a web-based ligand binding site prediction tool that employs the newest technologies. One of the goals of this thesis is to replace the outdated plugins for the structure and binding site visualization to keep the application relevant and up-to-date.

There are a few working web tools that can predict binding sites for the structure as well. The main issue with these tools is that most of them are utilizing outdated technologies and do not appear to be as intuitive as PrankWeb[3]. Moreover, PrankWeb focuses on the visual side of the prediction and provides a more detailed view of the results, while the other tools are relying on the user to interpret the downloadable results in other tools such as PyMOL [16].

We will cover a few of the existing web tools to provide a better understanding of some of the existing solutions for the ligand binding site prediction which are currently available.

1.5.1 IntFOLD

IntFOLD¹⁸ is a tool that may be used for predicting protein tertiary structures alongside disordered protein regions and ligand binding sites. This tool utilizes the

¹⁷More information about Prometheus is available at <https://prometheus.io/>.

¹⁸Available at <https://www.reading.ac.uk/bioinf/IntFOLD/>.

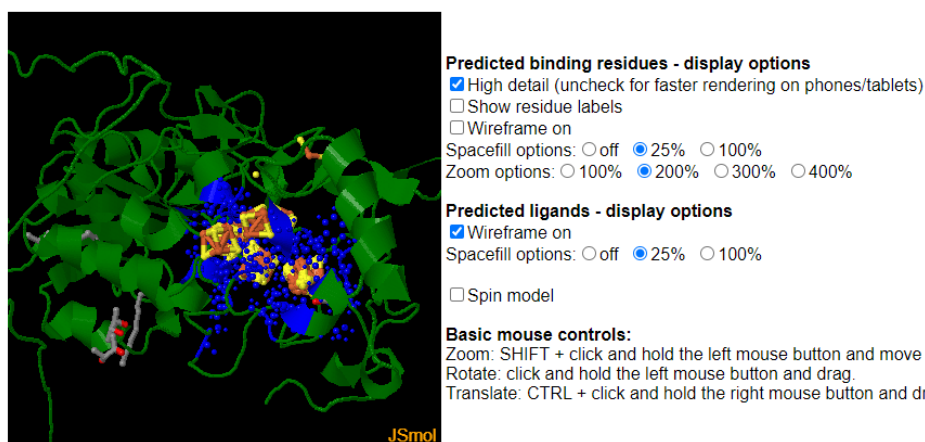


Figure 1.2 JSmol view of ligand binding residues prediction for T1114s2. Available at https://www.reading.ac.uk/bioinf/servlets/nFOLD/IntFOLD7results.jsp?time=17_8_50_25_11-5-2022_CASP_ALL_mesu41b7re376c81&md5=mesu41b7re376c81&targetname=T1114s2. The structure is shown in dark green, the binding sites prediction is shown in blue. Predicted ligands are shown yellow-orange. Available as a sample prediction from IntFOLD.

FunFOLD algorithm to create the predictions effectively [17]. IntFOLD appears to be still actively developed and is easily accessible to any potential user. The user interface is easy to use and requires only the protein sequence (FASTA) to be entered. On the other side, IntFOLD uses the JSMol library for visualizing the protein binding sites. JSMol is more of a simple viewer with fewer options than the LiteMol library used in PrankWeb. Both the visuals and the user interaction are limited. The user may modify the representation with a right mouse button click, but the interface is pretty complicated and non-intuitive. An example prediction is shown in Figure 1.2. Like PrankWeb, IntFOLD also provides a download option for the prediction results for PyMOL. The prediction takes a long time to be computed, typically around 24 hours.

1.5.2 COACH

COACH¹⁹ is a web-based tool designed specifically for binding site prediction, like PrankWeb. COACH uses a combination of substructure-comparison (TM-Site) and sequence-alignment (S-Site) methods for the computations of potential binding sites [18]. This combination of methods is generally slower than the machine learning method used in P2Rank, so the prediction once again is available to the user after a long time, typically around 24 hours. This tool allows the user not

¹⁹Available at <https://seq2fun.dcmf.med.umich.edu/COACH/>

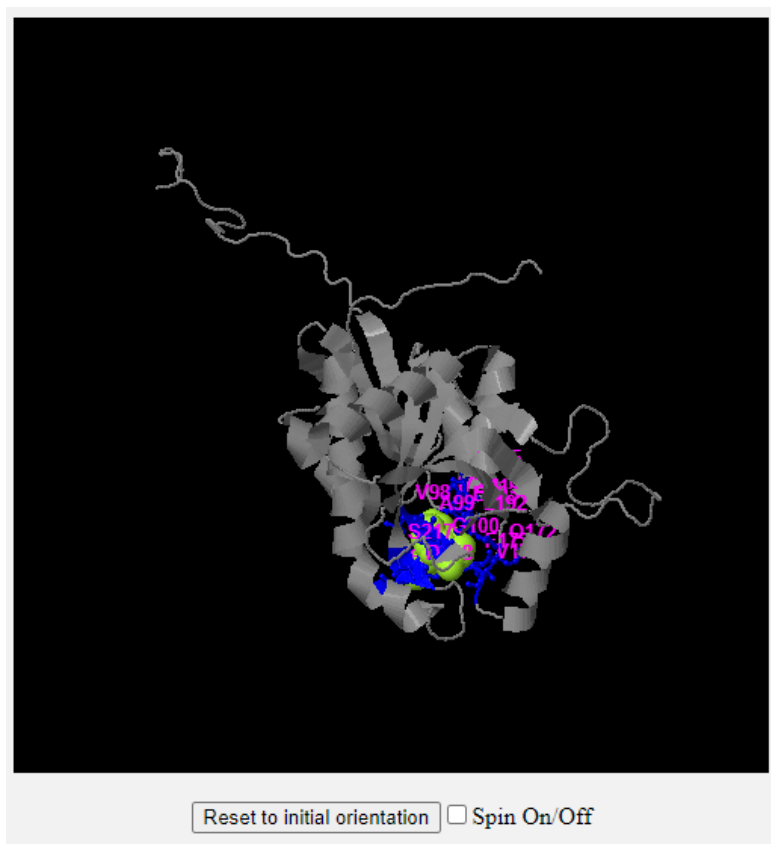


Figure 1.3 JSmol view of a COACH prediction result for a sample protein available at <https://seq2fun.dcmf.med.umich.edu/COACH/CH000001/>. Compared to IntFOLD, the structure has implicitly fewer details and the user may change the representation only with a right mouse button click via JSmol settings.

only to enter a FASTA sequence but also to upload or paste a PDB file. COACH allows the user to download the prediction results as well and does not focus on the very limited web visualization. An example prediction is shown in Figure 1.3.

1.5.3 DeepSite

DeepSite²⁰ is one of the newest tools for predicting binding sites. DeepSite is a part of the PlayMolecule framework that aims at the visual representation of the protein structures and their interactions in a user-friendly web interface [19] [20], which is highly applicable in computational drug discovery. DeepSite uses machine-learning-based methods to precisely predict the binding sites, which

²⁰Available at <https://www.playmolecule.com/deepsite/>

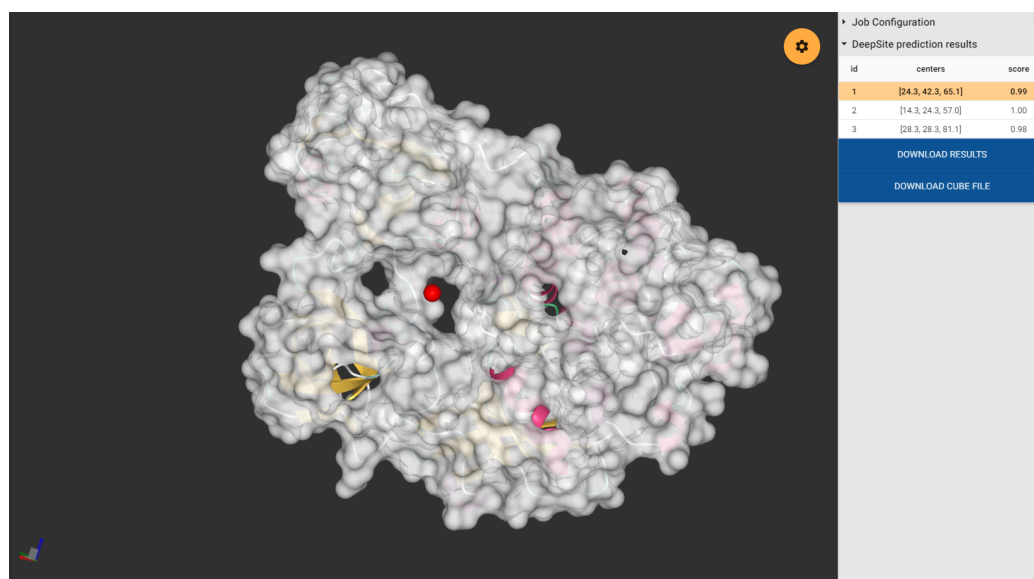


Figure 1.4 A DeepSite prediction for the 2SRC structure available at <https://www.playmolecule.com/deepsite/job/BD2ED307>. The surface representation is shown in white, underlying cartoon representation is colorful. The binding site prediction is shown by a red sphere.

makes the tool very fast [21]. The waiting times are significantly slower and the prediction is available to the user after a few minutes. The user may enter the PDB structure ID, which is a lot more convenient way to describe the structure than by entering the entire PDB file. On the other side, entering a custom format is a more generic way that does not limit the user to the existing PDB database. DeepSite utilizes the MolStar library for the results visualization and is similar to PrankWeb in terms of visual representation. On the other side, DeepSite provides the user only with a visualization of the center of the binding site and does not provide any information about the residues that are involved in the binding directly in the web viewer. The structure is shown in a surface representation. An example prediction is shown in Figure 1.4. The pros of this tool are definitely the speed of the prediction and an above-average visual representation. On the other side, the user is provided with little information about the binding site and may be used for a rather quick overview of the potential binding sites.

Chapter 2

Programming documentation

This chapter will introduce the reader to the code changes made to the original PrankWeb interface. The first part will focus on the frontend, the second part will focus on the plug-ins.

2.1 Frontend

The frontend of PrankWeb works as a TypeScript application. TypeScript is transpiled to JavaScript and bundled using Webpack. The application uses React for rendering a panel containing a toolbox (see figure 2.1), structure information and pocket data. Styles are provided by CSS files and SCSS Bootstrap. The application uses Material UI for a few of the component designs. All packages used in PrankWeb are installed using the npm tool. This architecture was already present in the original interface.

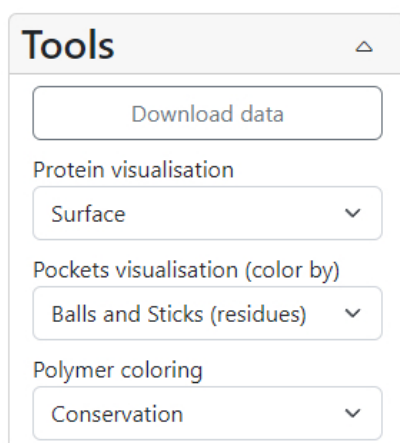


Figure 2.1 The toolbox component.

The former interface was based on the LiteMol library for visualizing the structure. LiteMol is no longer developed, so one of the goals was to replace this plug-in with a new, modern structure viewer from the same authors - MolStar. Not only have the visuals significantly improved, but the overall performance of MolStar is also much better [4].

Before the current change, PrankWeb used a different library for rendering the 1D sequence visualization. The original implementation used the Protael library for a simple visual representation of the pockets, binding sites and scores. This library is intended for creating customizable visualizations for a protein structure [22]. Protael is an old plug-in and in the original implementation, it needed to be modified in a significant way to fit the needs of PrankWeb. The new implementation presented in this thesis uses the RCSB Saguaro 1D Feature Viewer, which provides a more convenient way to display the pockets, binding sites and scores.

2.1.1 High-level overview

Firstly, let's describe the typical data flow between the frontend and backend. Assuming that this section works with the `frontend` folder of the repository, this folder contains all of the frontend logic including plug-in configuration files for Webpack and static assets. The static assets are located in the `public` subfolder. Those include several libraries (such as jQuery), CSS files and static images.

Moving onto the visualization logic to the `viewer` subfolder, the user starts at the `index.js` page, where they enter either a protein structure file or an RCSB protein identifier. A request via REST API is then sent from the frontend to the backend workers. If there is a free worker, the job is immediately processed. If there are no free workers, the job is queued and processed as soon as a worker becomes available. The backend workers then process the job and create a result file (see section 1.4.5 for details). Right after the request, the user is redirected to the `analyze.ts` file. The frontend periodically fetches both the status file and the log file to display the current job progress. When the job is finished, the user is redirected to the `viewer.ts` file. The entry point to the viewer is the `renderProteinView` method, which will be covered later on. The viewer file is responsible for visualizing the structure, the pockets, binding sites and scores. The last step is to combine the plug-ins so that the user can interact with the structure and the data.

An overview of the entire frontend architecture including is shown in figure 2.2.

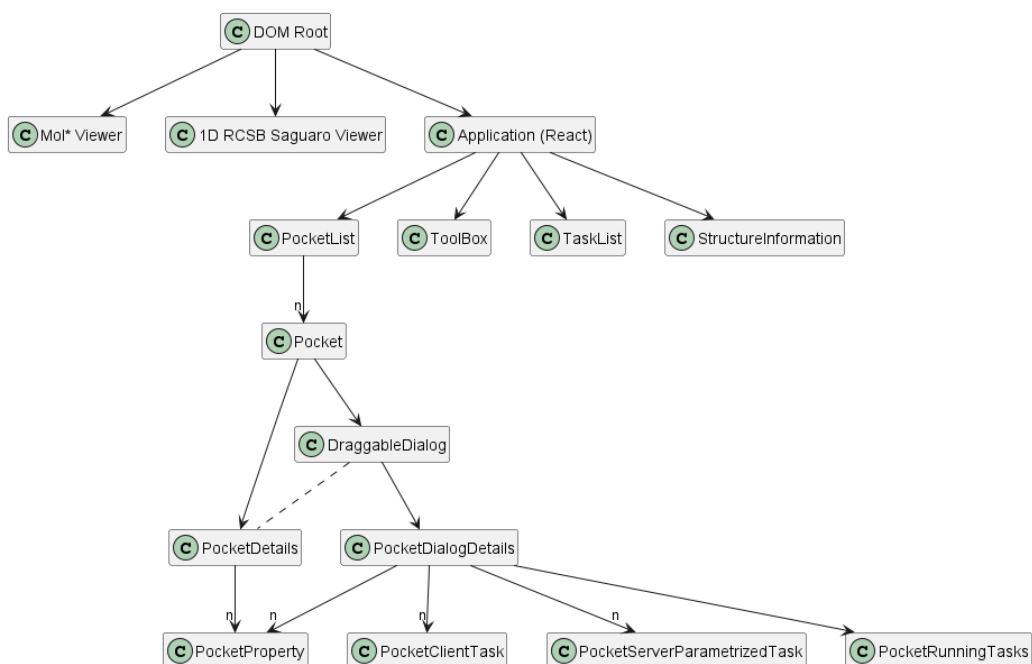


Figure 2.2 An UML diagram of the PrankWeb frontend architecture.

2.1.2 MolStar

MolStar (also Mol*) is a TypeScript library for visualizing protein structures. MolStar combines the strengths of the LiteMol [23] and NGL [24] libraries to provide a high-performance tool for bioinformatic scientists. The library is open-source and the code may be found on GitHub¹. One downside of MolStar is that it lacks detailed documentation. There are some examples available in the GitHub repository either directly in the source codes or in issues, but for more complicated code, the user may need to create their own issue and ask the developers directly. An example of the visualization is shown in figure 2.3.

After invoking the `renderProteinView` method, a MolStar viewer instance is created by calling the `createPluginUI` method from the library. An instance of `PluginUIContext` is returned. This instance is saved and used throughout the entire existence of the session. After the initialization, the main React component called `Application` is rendered for the first time. After mounting the component, the main visualization method `sendDataToPlugins` is called. This method from `data-loader.ts` is responsible for sending the prediction data for both plugins.

Some of the mentioned methods were already present in previous versions of PrankWeb. One of the goals of this thesis was to integrate MolStar into PrankWeb

¹<https://github.com/molstar/molstar>

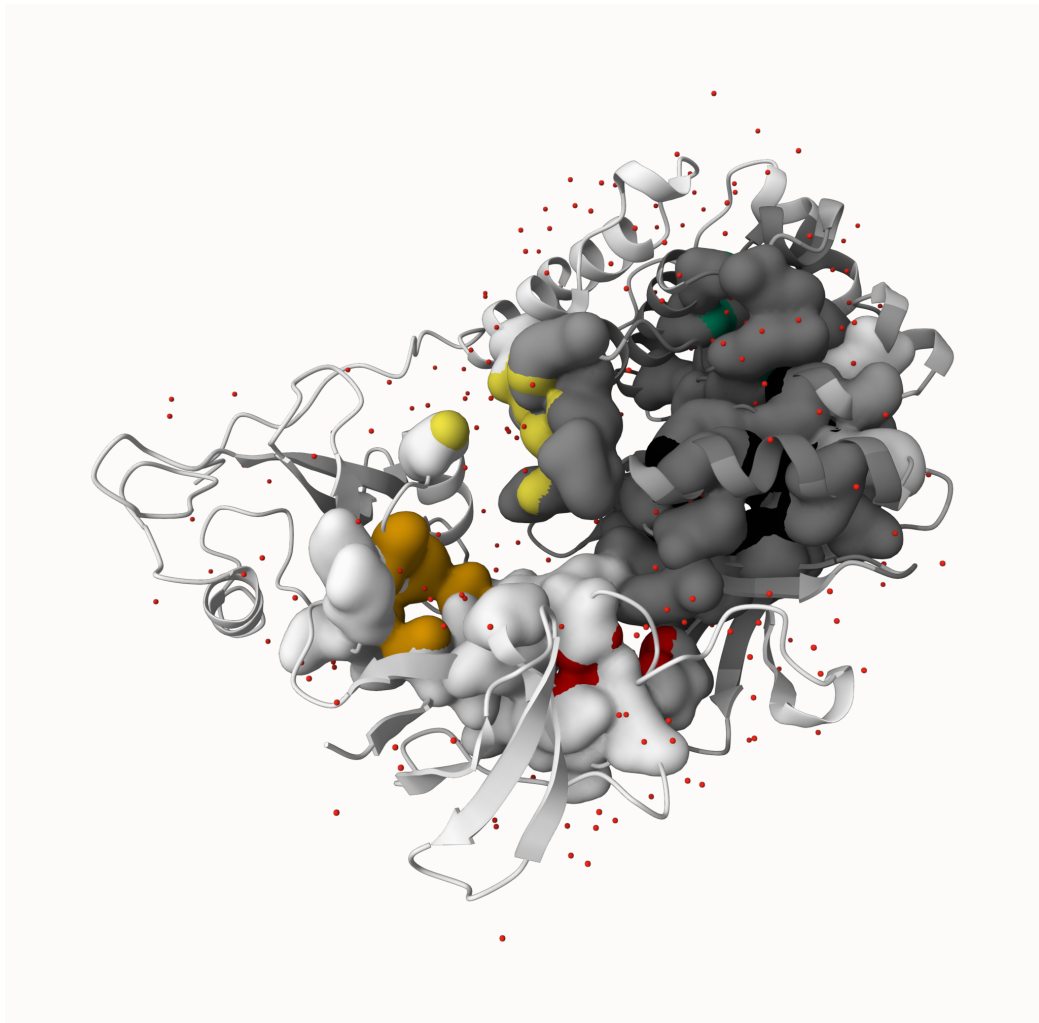


Figure 2.3 A screenshot of the MolStar viewer for the 2SRC structure. The structure is shown in cartoon representation and is colored by the conservation score. The pockets are shown in the surface representation.

by replacing LiteMol. This led to the removal of the original LiteMol-related code and introduced a few changes to the existing methods of loading data into the viewer. All of the MolStar-related code was written from scratch to fit the needs of PrankWeb. Most of the code related to MolStar is located in the `molstar-visualise.ts` file. The following text describes the flow of the MolStar-related interactions that create the core functionality of PrankWeb. These newly introduced methods call the MolStar library methods to create the visualization just from the data available in the prediction file and the structure file.

Firstly, the program asks for the API endpoint URL which resolves to a structure file. This file is loaded into MolStar via the `loadStructureIntoMolstar` method. This method parses the structure based on the file format and creates all of the available representations, such as surface, cartoon and ball-and-stick. It also tries to show the water molecules and ligands, if there are any.

When the structure is successfully shown to the user, a prediction is fetched from the API. Then, the 1D viewer is initialized. The 1D viewer connects to the MolStar plugin via callbacks. When the user hovers over a residue in the 1D viewer, the residue is highlighted in the MolStar plugin as well. When the user clicks a residue (or a pocket block), then the specific residue is focused in the MolStar plugin. The 1D viewer calls the `highlightInViewerAuthId` and `highlightInViewerLabelIdWithoutFocus` methods. See section 2.1.3 for more details about the code that explains the interactions between the 1D viewer and the MolStar plugin.

After initializing the viewer and visualizing the structure, we need to show the pockets as well. The current implementation uses the following procedure: process each of the pockets and create a custom colored representation that will be added to the visualization. This is done in the `createPocketsGroupFromJson` method, which simply invokes `createPocketFromJson` for each of the pockets. That method creates multiple representations to allow the user to decide between various ways to display the pocket. Currently, a ball-and-stick and surface representations colored by either surface atoms or the entire residues are available. In the future, more representations may be added. By default, only the surface atoms are shown as a pocket. The representations are added to a global variable containing all of them to allow switching between them based on user inputs from the React components.

For predicted structures, there is a special case. The predicted structure may contain areas that have not been properly analyzed, because no similar structures may be known. So, each of the residues is ranked with an AlphaFold pLDDT score that indicates how well a residue is predicted. Residues ranked with a score below 70 are ranked as low-confidence [7]. PrankWeb enables the user to hide these residues from the visualization. This is done by creating a second structure

visualization containing only the high-confidence residues. The user can switch between the two visualizations using a React component. For the creation of the second structure, we employ the `addPredictedPolymerRepresentation` method, which works in a similar way as creating the pockets.

After resolving the predicted structure, we have to ensure only the selected representations are visible to the user to ensure the best performance. This is done via the `showAllPocketsInRepresentation` method. Then, MolStar has to be linked to the 1D viewer, but in the opposite direction. Hovering over a residue in MolStar should highlight the corresponding residue in the 1D viewer. The method is called `linkMolstarToRcsb` and uses MolStar hover callbacks to achieve this behavior. The code is inspired by the MolArt library [25].

In the last step, it is necessary to compute average conservation (and potentially pLDDT) scores for each of the pockets. The JSON file provided by P2Rank contains the average scores only at the residue level, so the pocket-level average needs to be computed. As we assume that the pockets are not large, this computation is done in the frontend and is not cached in any way. This is done in the `computePocketConservationAndAFAverage` method.

In this step, the MolStar visualization is complete and ready to use. The user may interact with the structure either via the 1D viewer, the MolStar plugin, or the React components. The last part of the code is responsible for enabling interactions with the React tools. Currently, the user may interact with MolStar from the components in the following ways:

- Change structure representation
- Change pockets representation
- Color residues by conservation or pLDDT scores
- Hide low-confidence residues for predicted structures
- Hide/show all pockets
- Hide/show individual pockets
- Highlight a pocket including zoom

All of the interactions are handled by calling the respective methods directly from the React components, which are described in section 2.1.4. We will not cover details of the code, as it is mostly straightforward and the code is documented. The main idea is to apply transforms to the MolStar structure which is then re-rendered by the library.

In the end, we will show a brief code structure of the `molstar-visualise.ts` file containing the vast majority of MolStar-related code, just to give an overview of the MolStar integration.

Listing 2.1 A slightly edited version of a declaration file `molstar-visualise.d.ts`.

```
1 import { PluginUIContext } from 'molstar/lib/mol-plugin-ui/context';
2 import { PredictionData, PolymerColorType, PolymerViewType,
   PocketsViewType, Point3D } from '../custom-types';
3 import { RcsbFv } from '@rcsb/rcsb-saguaro';
4 import { StateObjectSelector } from 'molstar/lib/mol-state';
5 import { Expression } from 'molstar/lib/mol-script/language/expression';
6 /**
7  * Loads the structure to be predicted and adds the polymer
   representations to the viewer.
8  * @param plugin Mol* plugin
9  * @param structureUrl URL of the structure to be predicted
10 * @returns An array containing the model and structure.
11 */
12 export declare function loadStructureIntoMolstar(plugin: PluginUIContext,
   structureUrl: string): any;
13 /**
14 * Method used to show only the currently selected representation.
15 * @param value Currently shown type of polymer representation
16 * @param plugin Mol* plugin
17 * @param showConfidentResidues Whether to show only the confident
   residues
18 * @returns void
19 */
20 export declare function updatePolymerView(value: PolymerViewType, plugin:
   PluginUIContext, showConfidentResidues: boolean): void;
21 /**
22 * Method used to overpaint the currently selected polymer representation.
23 * @param value Currently shown type of polymer representation
24 * @param plugin Mol* plugin
25 * @param prediction Prediction data
26 * @returns void
27 */
28 export declare function overPaintPolymer(value: PolymerColorType, plugin:
   PluginUIContext, prediction: PredictionData): Promise<void>;
29 /**
30 * Method to create the pocket holder group (called "Pockets" in the tree)
31 * @param plugin Mol* plugin
32 * @param structure Mol* structure (returned from the first call of
   loadStructureIntoMolstar())
33 * @param groupName Group name (in this case "Pockets")
34 * @param prediction Prediction data
35 */
```

```

36 export declare function createPocketsGroupFromJson(plugin:
    PluginUIContext, structure: StateObjectSelector, groupName: string,
    prediction: PredictionData): Promise<void>;
37 /**
38  * Method which sets the visibility of one pocket in the desired
    representation
39  * @param plugin Mol* plugin
40  * @param representationType Type of the representation to be shown
41  * @param pocketIndex Index of the pocket
42  * @param isVisible Visibility of the pocket
43  * @returns void
44  */
45 export declare function showPocketInCurrentRepresentation(plugin:
    PluginUIContext, representationType: PocketsViewType, pocketIndex:
    number, isVisible: boolean): void;
46 /**
47  * Method which sets the visibility of all the pockets in the desired
    representation
48  * @param plugin Mol* plugin
49  * @param representationType Type of the representation to be shown
50  * @returns void
51  */
52 export declare function showAllPocketsInRepresentation(plugin:
    PluginUIContext, representationType: PocketsViewType): void;
53 /**
54  * Method which focuses on the residues loci specified by the user, can
    be called from anywhere
55  * @param plugin Mol* plugin
56  * @param chain Chain (letter) to be focused on
57  * @param ids
58  * @returns void
59  */
60 export declare function highlightInViewerLabelIdWithoutFocus(plugin:
    PluginUIContext, chain: string, ids: number[]): void;
61 /**
62  * Highlights the selected surface atoms, if toggled, the method will
    focus on them as well
63  * @param plugin Mol* plugin
64  * @param ids Surface atoms ids
65  * @param focus Focus on the surface atoms (if false, it will only
    highlight them)
66  * @returns void
67  */
68 export declare function highlightSurfaceAtomsInViewerLabelId(plugin:
    PluginUIContext, ids: string[], focus: boolean): void;
69 /**
70  * Method which adds predicted structure representation to the viewer
71  * @param plugin Mol* plugin
72  * @param prediction Prediction data

```

```

73 * @param structure Mol* structure (returned from the first call of
    loadStructureIntoMolstar())
74 * @returns void
75 */
76 export declare function addPredictedPolymerRepresentation(plugin:
    PluginUIContext, prediction: PredictionData, structure:
    StateObjectSelector): Promise<void>;
77 /**
78 * Method which gets selection of the confident residues (plddt > 70) for
    predicted structures
79 * @param prediction Prediction data
80 * @returns Expression with the selection of the confident residues
81 */
82 export declare function getConfidentResiduesFromPrediction(prediction:
    PredictionData): Expression;
83 /**
84 * Method which focuses on the loci specified by the user
85 * @param plugin Mol* plugin
86 * @param chain Chain (letter) to be focused on
87 * @param ids Residue ids
88 * @returns void
89 */
90 export declare function highlightInViewerAuthId(plugin: PluginUIContext,
    chain: string, ids: number[]): void;
91 /**
92 * Method which returns coordinates of the surface atoms
93 * @param plugin Mol* plugin
94 * @param ids Surface atom ids
95 * @returns An array of coordinates
96 */
97 export declare function getPocketAtomCoordinates(plugin: PluginUIContext,
    ids: string[]): Point3D[];
98 /**
99 * Method which connects Mol* viewer activity to the RCSB plugin
100 * @param plugin Mol* plugin
101 * @param predictionData Prediction data
102 * @param rcsbPlugin Rcsb plugin
103 * @returns void
104 */
105 export declare function linkMolstarToRcsb(plugin: PluginUIContext,
    predictionData: PredictionData, rcsbPlugin: RcsbFv): void;

```

2.1.3 RCSB Saguaro 1D Feature Viewer

Alongside the MolStar plugin, we use the RCSB Saguaro 1D Feature Viewer to display the pockets, actual binding sites and scores in a simple view mapped to the residues, which is the key feature of this plugin. The viewer is designed to

easily identify multiple annotations on a single residue.

The RCSB Saguaro 1D Feature Viewer is a TypeScript library for visualizing protein features in a 1D view. An example usage is shown in figure 2.4. The code is available on GitHub². Documentation is available and the usage is simple, but it is important to keep in mind that the library is still under active development and the interface may change in the future.

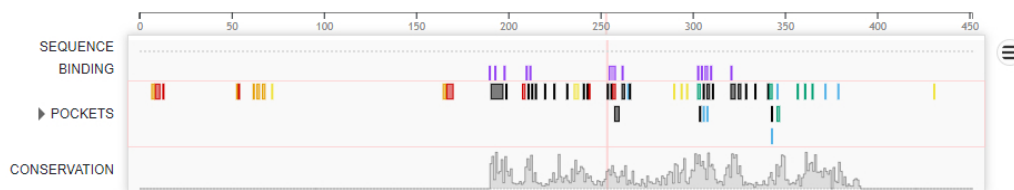


Figure 2.4 The RCSB Saguaro 1D Feature Viewer for the 2SRC structure.

In the original PrankWeb implementation, Protael was used for mapping the annotation to the single residues. In this thesis, we decided to use the RCSB Saguaro 1D Feature Viewer instead. The main reason for this decision is that the RCSB Saguaro 1D Feature Viewer is a more extensible library and is actively maintained. So, all of the code related to Protael was removed and replaced with new code using the RCSB Saguaro 1D Feature Viewer to provide the same functionality.

Most of the methods are defined in the `rscb-visualise.ts` file. In our case, as described in section 2.1.2, the 1D viewer gets initialized by the `sendDataToPlugins` method. This calls the `initRcsb` method which prepares all of the data for the plug-in.

The first thing to notice is calculating the viewer's actual width. This is done in the `calculateViewerWidth` method. Unluckily, the viewer allows only a fixed width measured in pixels. This is a problem, as the viewer is not responsive and every time the user resizes the window, the viewer would have to be re-initialized, which would be pretty inefficient. So, for the first time the page loads, we calculate the width of the viewer based on a knowledge of the desired width that is set via CSS. We also have to subtract a certain amount of pixels to account not only for the viewer itself but also for the margins, paddings and a custom viewer scrollbar, which is, once again, in-built in the viewer and is not customizable. We discussed this issue and concluded that the best solution is to add better-looking custom scrollbars to the viewer to provide a better user experience until the viewer is made responsive.

Then, the RCSB board is configured to interact with the MolStar viewer. This is done via the `onHighlight` and `elementClicked` methods. The first method

²<https://github.com/rcsb/rcsb-saguaro>

is called when the user hovers over a certain residue in the 1D viewer. The actual callback is debounced to prevent lagging. A request for highlighting the specific residue in MolStar is then sent. The second method is called when the user clicks on a residue in the 1D viewer. A similar request is sent to MolStar with the difference that the residue is also zoomed in.

The last step is to prepare the actual data for each of the so-called tracks. A track represents one row (or multiple rows) containing information about a specific annotation. This is done in the `createRowConfigDataRcsb` method. A sequence track simply contains the protein amino acids. The binding track contains the actual binding sites computed by P2Rank. Pocket tracks are distinguished by different colors, which are picked firstly based on color-blind schemes to ensure the best accessibility, and secondly randomly. The pocket color is changed in the original pocket data as well so that the same colors for the pockets are used throughout the application. The last tracks are the conservation and pLDDT tracks, if available. The computation of the pocket tracks is not straightforward, as the JSON file provided by P2Rank does not fit the viewer's interface, so the method highly relies on the JSON structure.

The 1D viewer does not rely on any React components and the only interaction may be done via the MolStar plugin simply by hovering over the residues, as described in section 2.1.2.

Similarly to MolStar, let's have a look at the `rcsb-visualise.ts` file structure and the most important methods.

Listing 2.2 A slightly edited version of a declaration file `rcsb-visualise.d.ts`.

```
1 import { RcsbFv, RcsbFvTrackDataElementInterface,
    RcsbFvRowConfigInterface } from '@rcsb/rcsb-saguaro';
2 import { PluginUIContext } from 'molstar/lib/mol-plugin-ui/context';
3 import { PredictionData } from '../custom-types';
4 /**
5  * Method which initializes the Rcsb viewer and adds the tracks to it.
6  * @param data Prediction data
7  * @param molstarPlugin Mol* plugin
8  * @returns The rendered Rcsb plugin.
9  */
10 export declare function initRcsb(data: PredictionData, molstarPlugin:
    PluginUIContext): RcsbFv;
11 /**
12  * Method to calculate the width of the viewer.
13  * @returns The width
14  */
15 declare function calculateViewerWidth(): number;
16 /**
17  * Method called when any element is clicked in the viewer.
18  * @param predictionData Prediction data
```

```

19 * @param molstarPlugin Mol* plugin
20 * @param trackData Data of the clicked track
21 * @param event Mouse event
22 * @returns void
23 */
24 declare function elementClicked(predictionData: PredictionData,
    molstarPlugin: PluginUIContext, trackData?:
    RcsbFvTrackDataElementInterface, event?: MouseEvent): void;
25 /**
26 * Method called when any element is highlighted in the viewer.
27 * @param data Prediction data
28 * @param molstarPlugin Mol* plugin
29 * @param trackData Data of the clicked track
30 * @param event Mouse event
31 * @returns void
32 */
33 declare function onHighlight(data: PredictionData, molstarPlugin:
    PluginUIContext, trackData: Array<RcsbFvTrackDataElementInterface>):
    void;
34 /**
35 * Method which creates all of the tracks for the Rcsb viewer.
36 * @param data Prediction data
37 * @returns Configuration for the viewer
38 */
39 declare function createRowConfigDataRcsb(data: PredictionData):
    RcsbFvRowConfigInterface<{}, {}, {}, {}>[];

```

2.1.4 React components

This section will describe the React components that are related to the structure visualization. Most of the components were already present in the previous version of PrankWeb, but there was a need to refactor them to make them more modular and reusable, as well as to add new features and improve the user experience.

The application uses the following components:

- **Application** - the main component of the app, responsible for the overall structure of the page including the viewers
- **ToolBox** - a component containing tools for changing the structure representation, coloring residues, hiding low-confidence residues and downloading data
- **StructureInformation** - a component containing the structure name

- **PocketList** - a component containing a list of all pockets with their scores and a button for hiding/showing all of them
- **Pocket** - a component containing a single pocket with its information and buttons for hiding/showing the pocket, highlighting the pocket and opening a dialog with the pocket details
- **PocketDetails** - a component defining the information of a Pocket component
- **PocketProperty** - a component visualizing a single property of a Pocket component

Other components will be covered in section 2.2 as they are related to the plug-ins.

All React components are defined in their own files in the `components` subdirectory. The components extend the `React.Component` class to utilize the props and state features of React. The components invoke the responsible RCSB and MolStar methods on various value changes, clicks on buttons and other events. Some of the props get passed from the parent component to the children, which allows easy communication between them and enforces best practices and minimizes code duplication. Some of the components have state variables as well that mostly indicate their current visibility.

The components are styled as Bootstrap cards and thus they are responsive. Although we expect the user to use PrankWeb mainly on a desktop computer, it is necessary to provide a good first experience when encountering the application on a smaller screen.

2.2 Plug-ins

The second goal of this thesis was to introduce the possibility to extend the current functionality of the application by adding new plug-ins (also called tasks, these terms are used interchangeably). The plug-ins enable the postprocessing of the pockets.

After a discussion, we created the following plug-in categories: client-side and server-side. It is assumed that the client-side plug-ins will be used for simpler tasks that may be computed from the existing data in each session, on the contrary, the server-side plug-ins will be used for complex tasks that are more time-consuming and possibly utilize third-party software.

The PrankWeb interface for both of the tasks works as a set of React components that are modular, so new plug-ins may be introduced easily without modifying much of the current code.

For easy access to plug-ins, we have created a possibility to view the pocket details not only directly in the pocket list but in a draggable dialog as well. This dialog is defined in the `DraggableDialog` and `PocketDialogDetails` classes that provide the implementation. The user may open multiple dialogs at once as well as drag them around the screen. The dialogs allow interaction behind them, so the user may still manipulate both of the viewers.

This decision was discussed more and multiple designs were proposed and tested. All of the possible variants are listed in appendix A.4. The final decision was made based on the fact that a vast majority of users are expected to use a big-screen device, so to keep the original usage easy and intuitive, we decided to keep the original design. One downside of this implementation is that the dialogs are not intended to be used on mobile devices. In other words, the dialogs are not available for small screens which makes this section irrelevant for those devices. On the other side, we believe that this is a reasonable trade-off.

In the following subsections, we will introduce the general plug-in interface and the implemented plug-ins.

2.2.1 Client-side plug-ins

The main intention of client-side plug-ins is to create a possibility to enable simple post-processing from the prediction data directly in the application. The tasks computed on the client-side stay in the current session.

Currently, the client-side plug-ins are defined in a `PocketDialogDetails` component as they are dialog-specific. A new interface for these plug-ins was created in the `PocketClientTask` component. Let's have a look at the interface.

Listing 2.3 An edited version of the `pocket-client-task.tsx` component.

```
1 import React from "react";
2 import { ClientTaskData, ClientTaskType, PocketData } from
  '../..../custom-types';
3 import { PluginUIContext } from "molstar/lib/mol-plugin-ui/context";
4 import { PredictionInfo } from "../prankweb-api";

6 export default class PocketClientTask extends React.Component
7   <{
8     title: string,
9     inDialog: boolean,
10    pocket: PocketData,
11    plugin: PluginUIContext,
12    taskType: ClientTaskType,
13    prediction: PredictionInfo,
14    compute: () => Promise<ClientTaskData>,
15    renderOnComplete: (data: ClientTaskData) => JSX.Element
```



```
16   }, {
17     taskData: ClientTaskData | undefined,
18     computed: boolean,
19     loading: boolean
20   }> {
21     //...
22   }
```

Some of the props and state variables are self-explanatory, and some of them are passed from the parent component. We will focus on the client-side specific props and state variables that define their behavior.

Firstly, let's take a look at the `taskType` prop. This prop defines the type of task that is defined in the `ClientTaskType` enum. This enables a possible behavior difference of the client-side plug-ins based on their type. The next prop is a method called `compute`. This is the key method of the plug-in that provides the actual post-processing of the prediction data. The behavior is defined by the parent component. In the plug-in definition, the `compute` method is called. We expect this method to return a promise containing data conforming to the `ClientTaskData` interface. After receiving the result, the result needs to be rendered. During this entire process, state variables `taskData`, `computed` and `loading` are used to indicate the current computation progress.

Originally, one implementation for showing the result was created, but to provide a more generic approach to the problem, a second method was introduced to the props - the `renderOnComplete` method. This method takes the computed data as a parameter and returns a JSX component that renders the data inside the original dialog.

Currently, the client-side plug-ins are defined in the `tasks` subdirectory. The actual implementations are `.tsx` files and should contain both of the methods for the computation and rendering to keep the code clean.

As an example of a client-side plug-in, we have implemented a plug-in that computes the expected volume of the pocket. For this purpose, we have created a `tasks/client-atoms-volume.tsx` file that contains `computePocketVolume` and `renderOnTaskVolumeCompleted` methods that conform to the `compute` and `renderOnComplete` props of the `PocketClientTask` component. In this post-processing, we are using a convex hull algorithm to get the coordinates of the hull vertices. Then, we use a simple formula to compute the volume of the convex hull using the coordinates of the vertices based on triangulation of the hull [26]. The result is then rendered in the dialog as a simple number. This implementation saves the computed results in a hashmap directly in the implementation file. This is not the most efficient way to store the results, on the other hand, it is a simple solution that is sufficient for this use case.

Creating a custom new client-side plug-in is covered in detail in section 3.3.

2.2.2 Server-side plug-ins

Server-side plug-ins create an opportunity to perform more complex postprocessing that takes more time to compute. We intended to allow users to use third-party software for postprocessing. Server-side tasks are meant to be parametrized by the user in some way.

The plug-ins are defined in a similar way as the P2Rank executor, which may be processed by Docker containers (as mentioned in section 1.4).

There are a few steps that have to be done to implement a new server-side plug-in. First, API endpoints for the plug-in, as well as the folder and file structure, need to be defined. This is done in the `web-server` folder in Flask. Usually, it is expected that Flask will prepare the needed directories and files for the actual Docker container. For this process, it is typical to create a new Python class to contain all of this logic.

Then, in the Celery client configuration file in Flask, new bindings are created for the new plug-in. This includes creating a new Celery queue as well as introducing a new task that gets executed on the backend identified by its name.

After completing all of the Flask tasks, a new Docker container has to be created. The container should install Python with Celery and any other tools needed for the post-processing. It is expected that a new `Dockerfile` will be created for this purpose. After setting up the container, Celery bindings have to be created in a similar way as in Flask. Then, any method may be called from this Celery binding. This means that the actual post-processing may be finally done here. The first option is to do this in Python entirely, but it is also possible to call any other tool from the container. The only requirement is to have everything set up properly.

The Docker container is also responsible for saving the results of the post-processing. This means that either some of the Python files or the third-party tool itself has to save the results in the correct location. Keep in mind that the location is very likely to depend on the API endpoints defined in Flask. Now, the Docker container is ready to be used.

It is highly recommended to include the newly created container in the `Docker-compose` file. This makes the deployment easy and allows simpler testing of the new plug-in³.

After completing all of the steps, the new plug-in should be ready to be used. The only thing left is to include the plug-in to the frontend. For the frontend interaction, a new React component was introduced.

`PocketServerParametrizedTask` is meant to be used for the server-side plug-ins that require some parameters to be set by the user. This component is

³With Docker-compose, it is possible to restart just one of the containers, which makes the debugging faster.

used in `PocketDialogDetails` similarly to client plug-ins (more about dialog in section 2.2.1). Let's have a look at the server plug-in component.

Listing 2.4 An edited version of the `pocket-server-parametrized-task.tsx` component.

```
1 import React from "react";
2 import { PocketData, ServerTaskData, ServerTaskType } from
  '../..../custom-types';
3 import { PluginUIContext } from "molstar/lib/mol-plugin-ui/context";
4 import { PredictionInfo } from "../..../prankweb-api";

6 export default class PocketServerParametrizedTask extends React.Component
7   <{
8     title: string,
9     inDialog: boolean,
10    pocket: PocketData,
11    plugin: PluginUIContext,
12    taskType: ServerTaskType,
13    prediction: PredictionInfo,
14    serverTasks: ServerTaskData[],
15    modalDescription: string,
16    compute: (hash: string) => Promise<any>
17    renderOnComplete: (responseData: ServerTaskData, hash: string) =>
      JSX.Element
18    hashMethod: (prediction: PredictionInfo, pocket: PocketData,
      formData: string) => string
19  }, {
20    taskData: ServerTaskData | undefined,
21    computed: boolean,
22    loading: boolean,
23    modalOpen: boolean,
24    formData: string,
25    hash: string
26  }> {
27    //...
28  }
```

The component works similarly to the client-side plug-in component. There are two methods that are responsible for the computation and rendering of the results. One of the differences is that these two methods are now identifiable by a hash that has to be computed from the parameters before the actual computation. The computation of a hash is done via the `hashMethod` method. This hash is not only used for frontend identification but also for the backend. It is up to the developer, whether they will consider it as a possible input or just as a string for identification. This technique allows us to differentiate between multiple tasks of the same type with different parameters.

There are more state variables as well. `modalOpen` and `formData` were introduced for receiving input from the user, the current implementation opens a modal dialog with a text area. The hash is then computed from the input stored in the `formData` variable.

As the task results are persistent on the server, a new variable called `serverTasks` was introduced in the main component of the app. This component then periodically checks for all computed tasks for the given structure. The objects in this array are mutable and are modified for the purpose of the frontend rendering, so only the response data for tasks completed in this session are saved.

This allows us to render the hashes of the tasks that were computed both in this session and in previous ones. For this purpose, a new component called `TaskList` was introduced to display the list of tasks.

One last change was made to the `PocketDialogDetails` component. A new component `PocketRunningTasks` was created to display a list of tasks that are either running or completed in this session. This is done to make sure that the task results are available to the user in case the dialog is closed and reopened. Moreover, this allows the user to enter multiple parameters for the same task and see the results for all of them.

One of the goals of this thesis was to implement basic **server-side molecular docking**. Docking may be computationally very demanding, which makes it a perfect candidate for a server-side plug-in.

Firstly, we designed a public API for docking requests. There were three main needs:

1. Starting a new docking task with a given user input.
2. Receiving task results.
3. Receiving information about all tasks (including their status).

The API routes were added to the Flask configuration file. The routes are defined as follows:

Listing 2.5 An edited version of the `api_v2.py` file.

```
1 from .docking_task import DockingTask
2 from flask import Blueprint, request
3 api_v2 = Blueprint("api_v2", __name__)
4 # ...
5 @api_v2.route(
6     "/docking/<database_name>/<prediction_name>/post",
7     methods=["POST"]
8 )
9 def route_post_docking_file(database_name: str, prediction_name: str):
10     data = request.get_json(force=True) or {}
```

```

11     dt = DockingTask(database_name=database_name)
12     return dt.post_task(prediction_name.upper(), data)

14 @api_v2.route(
15     "/docking/<database_name>/<prediction_name>/public/<file_name>",
16     methods=["POST"]
17 )
18 def route_get_docking_file_with_param(database_name: str,
19     prediction_name: str, file_name: str):
20     data = request.get_json(force=True)
21     param = data.get("hash", None)
22     if data is None or param is None:
23         return "", 404
24     dt = DockingTask(database_name=database_name)
25     return dt.get_file_with_post_param(prediction_name.upper(),
26         file_name, param)

26 @api_v2.route(
27     "/docking/<database_name>/<prediction_name>/tasks",
28     methods=["GET"]
29 )
30 def route_get_all_docking_tasks(database_name: str, prediction_name: str):
31     dt = DockingTask(database_name=database_name)
32     return dt.get_all_tasks(prediction_name.upper())

```

After designing the routes, a new class called DockingTask was created. DockingTask class is responsible for serving the API requests. This includes the definition of the directory structure, saving the input files, preparing the info file, and calling the actual docking tool on the backend. Most of the file errors are handled in this class, as there are many opportunities for wrong inputs and requests. As all of the methods are important, we will focus on the one that is responsible for the actual docking. That is the `submit_directory_for_docking` defined in the Celery client configuration file. This method sends a task named docking to the Celery queue. Note that for docking, a new Celery queue was created in the configuration file.

Listing 2.6 An edited version of the `celery_client.py` file.

```

1 import os
2 import celery
3 prankweb = celery.Celery("prankweb")
4 prankweb.conf.update({
5     "task_routes": {
6         # the key is the name of the task, the value is the name of the
7         # queue
8         'prediction': 'p2rank',
9         'docking': 'docking',

```

```

9     }
10  })
11  #...
12  def submit_directory_for_execution(directory: str):
13      prankweb.send_task("prediction", args=[directory])

15  def submit_directory_for_docking(directory: str, taskId: int):
16      prankweb.send_task("docking", args=[directory, taskId])

```

Then, a new Docker container was created in the `executor-docking` directory and added to the Docker-compose files. We introduced a new Docker volume to preserve the results as well. An important part of this step is to mark this container with the right Celery queue in the command.

Listing 2.7 An edited extract from the `docker-compose.yml` file.

```

1  executor-docking:
2      build:
3          context: ./
4          dockerfile: ./executor-docking/Dockerfile
5          args:
6              UID: ${UID}
7              GID: ${GID}
8      command: ["celery", "--app=celery_docking", "worker",
9              "--queues=docking", "--concurrency=4",
10             "--hostname=executor-docking"]
11     restart: unless-stopped
12     volumes:
13         - docking:/data/prankweb/docking
14         - predictions:/data/prankweb/predictions
15     depends_on:
16         rabbitmq:
17             condition: service_healthy

```

The newly created Docker container is described in the `Dockerfile`. All of the needed tools for docking are downloaded and set up there. After setting up the container, Celery bindings are created similarly as in Flask in the `celery_docking.py` file. This file contains routing for the Celery task sent from the Flask server. This calls methods from `run_task.py` that contains that handle all of the work with the info file for the given structure (updates the status and timestamps). From there, the `dock_molecule` method is called. `dock_molecule` is responsible for the actual docking.

Firstly, the structure is unzipped from the gzipped file created during the initial prediction. Then, we treat `.pdb` files a bit differently, structures in this file format are cleaned using the `lePro` tool to remove unnecessary information. After the possible structure cleanup, a tool called `prepare_receptor` from the ADFR

Suite to prepare the structure for docking [27]. Moving on, ligand preparation is done using the `rdkit` and `meekeo` Python packages [28]. The last step before docking is to prepare the bounding box for the docking program. This is done manually by parsing the input file, saving the result to a text file.

In this stage, everything is properly set up for the docking. We use the `AutoDock Vina` [29] program for the docking. `AutoDock Vina` enables us to dock the ligand in a simple and fast way, on the other side, the results are not as accurate. We still decided to use this tool due to a rather simple setup and fast computation. The docking program is invoked with the prepared parameters and the results are saved to a public folder.

After the docking, the info file is updated with the result. The output file then may be downloaded by the actual user. The file may be viewed in several tools, such as `PyMOL`.

Now, the only thing left is to implement the frontend components to provide a user-friendly way to dock the molecules.

We created a `ServerTaskData` enum for the potential to introduce new server-side plug-ins. Alongside the enum, the `ServerTaskDataContents` interface for data from the server side. The main logic of server-side plugins is defined by implementing the `PocketServerParametrizedTask` component that was added to the `PocketDialogDetails` component in a similar way to the client plug-ins (see section 2.2.1). The implementation was done in the `tasks/server-docking-task.tsx` file. This file has private methods that compute a bounding box for docking before submitting the task, and public exported methods that are responsible for calling the API and displaying the results. One of the methods could hash the input, but as long as we are using SMILES molecule representation [30], there is no need to hash the input.

To finish the frontend implementation, a few more components were updated in order to show all results correctly. Firstly, we modified the `Application` component to keep all of the docking tasks in their state variable. This is done in the `getTaskList` method to create an ability to show all results of the completed tasks in the current session. This state variable called `serverTasks` then propagates to other components. Lastly, we bound the previously created methods to the `PocketRunningTasks` component to display the results of the docking tasks in the dialog window.

Creating a custom new server-side plug-in is briefly described in section 3.3.

Chapter 3

User documentation

In this chapter, the user documentation is presented. Firstly, the deployment process is described. We distinguish between two types of users, regular users and developers. We expect regular users to use PrankWeb without modifying the existing code. On the other side, we expect developers to add their own plug-ins or make other changes to the existing architecture. For this purpose, we describe a guide for both types of users.

3.1 Deployment

This section describes how to deploy the application. The preferred way is to deploy PrankWeb using Docker.

3.1.1 Docker deployment

Firstly, Docker and Docker Compose need to be installed¹.

After installing Docker, the PrankWeb repository needs to be cloned, preferably using Git².

1. Create a directory for the PrankWeb repository and navigate to it.
2. Clone the repository using the following command:

```
1 git clone https://github.com/cusbg/prankweb.git .
```

3. Create directories for each of the volumes.

¹Downloadable from the official website at <https://docs.docker.com/get-docker/>

²Downloadable from the official website at <https://git-scm.com/downloads>

4. Create mounts for each of the volumes in the `docker-compose.yml` file. The mounts are created as follows, simply update the `/tmp/` paths to the paths of the directories created in the previous step:

```
1  docker volume create --name prankweb_rabbitmq --opt type=none --
    opt device=/tmp/rabbitmq --opt o=bind

3  docker volume create --name prankweb_conservation --opt
    type=none --opt device=/tmp/conservation --opt o=bind

5  docker volume create --name prankweb_predictions --opt type=none
    --opt device=/tmp/predictions --opt o=bind

7  docker volume create --name prankweb_services --opt type=none --
    opt device=/tmp/services --opt o=bind

9  docker volume create --name prankweb_docking --opt type=none --
    opt device=/tmp/docking --opt o=bind
```

5. Optionally, create a `.env` file to change the default variables defined in the `docker-compose.yml` file. Notice that UID and GID need to have the write permissions to the directories created in the previous steps.
6. Build the Docker images using the following command:

```
1  docker-compose build
```

7. Optionally, download the conservation database (keep in mind that this database is large, around 30 GB):

```
1  docker-compose run --rm executor python3
    /opt/hmm-based-conservation/download_database.py
```

8. Start the containers using the following command:

```
1  docker-compose up
```

The application is now accessible at `http://localhost:8020/`.

Although the main deployment process will not change much, the best way to get the current information is to refer to the official documentation at `https://github.com/cusbg/p2rank-framework/wiki/PrankWeb-deploy-with-Docker`.

3.1.2 Local deployment

Local deployment is not recommended, as the support for some components is limited.

The only component that is developed better locally is the frontend (when not considering backend API changes). To run the frontend locally, firstly clone the repository as described in 3.1.1. Then, navigate to the frontend directory.

It is recommended to have a look at the `server/configuration.js` file to set up the proxy server that will serve potential calls. The proxy service should be running, it is possible to use the default PrankWeb website at `https://prankweb.cz`.

This is the default configuration:

```
1 // This configuration is used only in develop mode.
2 module.exports = {
3   // Port used to run-develop instance.
4   "port": 8075,
5   // Use this to server data from files. Thus, you can develop
6   // frontend without the need to run another component.
7   //"proxy-directory": "../data/database/",
8   // Use the option below to proxy commands to the task runner
9   // instance.
10  // This allows you to run tasks or connect to an existing instance
11  // (https://prankweb.cz).
12  "proxy-service": "https://prankweb.cz",
13 }
```

After setting up the proxy server, the frontend npm modules need to be installed. This can be done using the following command:

```
1 npm ci
```

Then, it is possible to run the frontend using the following command:

```
1 npm run dev
```

The frontend is now accessible at `http://localhost:8075/` (with the port specified in the configuration file).

Some of the tools from the executors like P2Rank may work from the command line as used in the Python scripts or Dockerfiles, for more information refer to those.

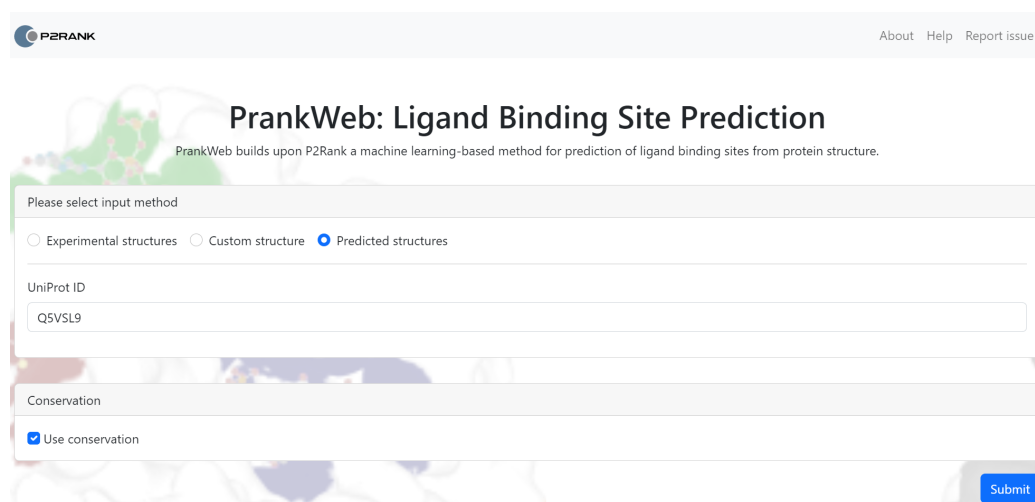
Official documentation for the local deployment is available at <https://github.com/cusbg/p2rank-framework/wiki/PrankWeb-deploy-for-development>.

3.2 User guide

Regular users are not expected to modify the existing code. The following section describes how to use the application.

The only requirement is to have a web browser with JavaScript and WebGL support.

Firstly, select a protein structure for the analysis in an input form. Expected inputs are either a protein structure code for experimental structures from the RCSB PDB, a custom PDB or mmCIF file, or a UniProt ID of a predicted structure. It is possible to use conservation data from the HMM-based conservation database to provide more information for the prediction. For experimental structures, it is possible to restrict the prediction just to specified chains. The input form is shown in figure 3.1.



The screenshot shows the PrankWeb application interface. At the top left is the P2RANK logo, and at the top right are links for 'About', 'Help', and 'Report issue'. The main heading is 'PrankWeb: Ligand Binding Site Prediction', with a subtext: 'PrankWeb builds upon P2Rank a machine learning-based method for prediction of ligand binding sites from protein structure.' Below this is a form titled 'Please select input method' with three radio buttons: 'Experimental structures', 'Custom structure', and 'Predicted structures' (which is selected). Underneath is a text input field for 'UniProt ID' containing the value 'Q5VSL9'. A second section titled 'Conservation' has a checked checkbox for 'Use conservation'. A blue 'Submit' button is located at the bottom right of the form.

Figure 3.1 Input form for the PrankWeb application.

Then, a prediction task is created and sent to the backend. During the prediction, a log of the task is shown. After the task finishes, the viewers are shown. The PrankWeb viewers are shown in figure 3.2.

Once the protein visualization is loaded, three main panels appear - sequence visualization, structural visualization and the pocket panel.

On the left side, both viewers visualizing the structure are shown. There is the RCSB 1D Saguaro Viewer on the top for the 1D visualization of the properties of the structure and predicted binding sites. On the bottom, we may see the Mol* viewer for the 3D visualization of the structure and pockets.

By default, the protein surface is displayed, and individual pocket areas are highlighted with different colors. Ligands may be displayed as separate molecules as well, if available. It is possible to change the colors of the protein based on the

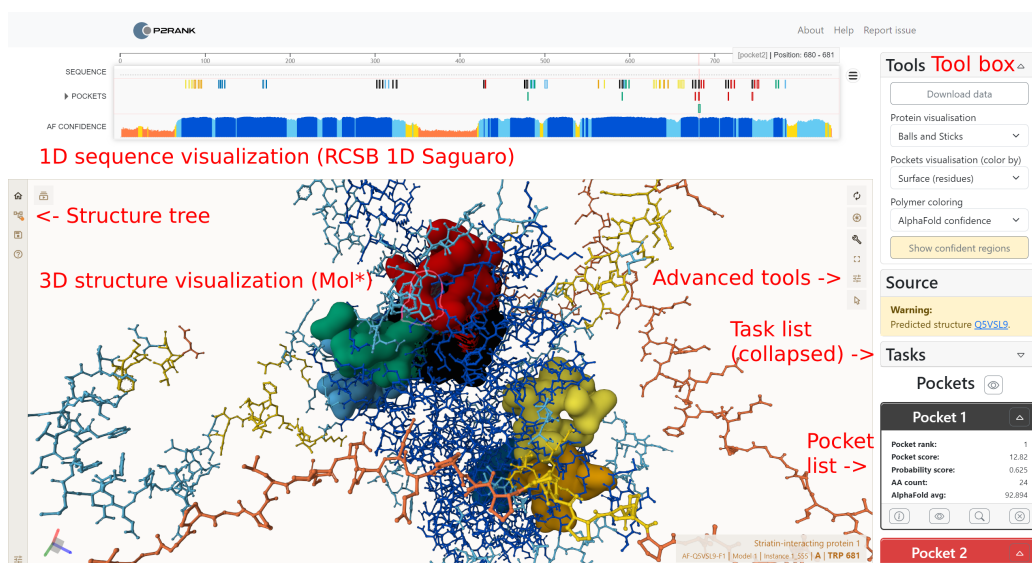


Figure 3.2 The main PrankWeb interface showing visualizations of the P21802 UniProt structure.

conservation or pLDDT score in the toolbox on the right side. For conservation coloring, darker residues depict a higher score. For pLDDT coloring, the colors are defined by the AlphaFold confidence score [14].

The 3D structure may be rotated by dragging the mouse while holding the left mouse button. The zoom is controlled by the mouse wheel or by pinching on touch devices. For moving the protein structure, the right mouse button may be used. The structure may be reset to the default position by clicking the reset button in the top right corner of the 3D viewer.

Mol* provides more features for visualization. Using the buttons in the top-right corner, one can:

- Reset the camera.
- Create a snapshot of the current visualization.
- Toggle the advanced control panel.
- Toggle full-screen mode.
- Setup the scene such as the visualization background or the field of view.
- Toggle the selection mode.

After toggling the advanced control panel, the following may be done:

- Work with the structure and download it.
- Toggle the state tree and thus toggle multiple of the available representations.
- Save the current plugin state.
- View the help panel.

For more information, please refer to the Mol* documentation at the official website³.

The RCSB 1D Saguaro Viewer displays the protein sequence. We implemented the functionality in a way the viewer does the following:

- All chains are concatenated into a single sequence and shown at once.
- Colored rectangles depict the predicted binding sites. The colors are the same as in the 3D viewer.
- Real binding sites are shown as well. As real binding sites, we consider any residues within 4 Å from a ligand atom.
- If available, conservation and pI-DDT scores are shown as well.

The Mol* viewer and the 1D Saguaro Viewer are synchronized. This means that when hovering over a residue in the 1D viewer, the corresponding residue in Mol* is highlighted. The same applies to the 3D viewer. When hovering over a residue in Mol*, the corresponding residue in the 1D viewer is highlighted. Furthermore, after clicking on a residue in the 1D viewer, Mol* focuses on the corresponding residue.

The right panel consists of a toolbox component, a structure information component, a task list component and a pocket list component.

The toolbox component allows a change in the coloring of the protein structure as mentioned before, and to download information about the prediction.

The structure information component simply shows the structure code.

The task list component shows the list of all backend tasks that have been created (i.e. docking tasks).

The pocket list component shows the list of all pockets in the structure. The pockets are sorted by their probability score. The pockets contain several buttons for the following interactions:

- Display details about the pocket.

³The Mol* documentation is available at <https://molstar.org/viewer-docs/>

- Show only the selected pocket.
- Focus on the pocket in Mol*.
- Hide the pocket.

The details about the pocket are shown in a modal dialog. From the dialog, it is possible to run the client-side and server-side tasks. The dialog window is shown in figure 3.3.

The screenshot shows a modal dialog titled "Pocket 1" with a dark header. The content is as follows:

Pocket rank:	1	
Pocket score:	53.44	
Probability score:	0.983	
AA count:	43	
Residues:	A_133, A_134, A_135, A_136, A_138, A_280, A_282, A_283, A_284, A_285, A_286, A_287, A_292, A_339, A_341, A_377, A_378, A_383, A_384, A_385, A_455, A_484, A_491, A_567, A_568, A_569, A_574, A_648, A_649, A_650, A_657, A_665, A_667, A_672, A_673, A_674, A_675, A_676, A_677, A_678, A_680, A_88, A_90	
Total atoms volume (Å ³):		COMPUTE
Total docking tasks:		COMPUTE
Docking task:		COMPUTE
Docking task (CN1CCC(C(C1)O)C...):		RESULT

CLOSE

Figure 3.3 The pocket details dialog component.

3.3 Developer

Developers may modify the existing code to their needs. They may edit any of the existing containers described in section 1.4. One of the expected changes is to implement custom client-side and server-side plug-ins. The following sections describe how to do that.

Before reading the contents of this section, please familiarize yourself with the topic of plug-ins described in section 2.2.

3.3.1 Client-side plug-ins

Implementing custom client-side plug-ins is simple. The creation may be simplified into the following steps:

1. Introduce a new task type in the `frontend/custom-types.ts` file by adding a new value to the `ClientTaskType` enum.
2. Create a new `.tsx` file, preferably in the `frontend/tasks` directory.
3. In this file, implement a method that will compute the task and returns a `Promise<ClientTaskData>`.
4. Implement a method that will render the completed task in the dialog window. This method takes the `ClientTaskData` as an argument and returns a React component - `JSX.Element`. An example is shown in listing 3.1.
5. Optionally, implement saving the computed volumes at least to a hashmap or implement another sort of caching.
6. Add the new task component anywhere to be rendered. We suggest adding it to the existing dialog window represented by the `PocketDialogDetails` component. An example is shown in listing 3.2.

Listing 3.1 An example client-side task (in this case referred to as `frontend/tasks/client-sample-task.tsx`).

```
1 import React from "react";
2 import { ClientTaskData, ClientTaskType } from "../custom-types";

4 export async function computeTask(params: any): Promise<ClientTaskData> {
5     //Do some computation here...
6     return {
7         "data": 42069, //computed value (any type)
8         "type": ClientTaskType.SampleTaskCount
9     }
10 };

12 export function renderTask(data: ClientTaskData): JSX.Element {
13     return <span style={{float: "right", marginLeft:
14         "1rem"}}>{data.data}</span>;
15 }
```

After this implementation, the client-side plug-in is ready to be used.

Listing 3.2 A modified `PocketDialogDetails` component including the newly introduced client-side plug-in (`frontend/viewer/components/pocket-dialog-details.tsx`).

```
1 import React from "react";

3 import PocketClientTask from "../pocket-client-task";
4 import { ClientTaskType } from "../../custom-types";
5 import { computeTask, renderTask } from "../../tasks/client-sample-task";

7 export default class PocketDialogDetails extends React.Component
8 <
9 //...
10 >
11 {
12 //...
13 render() {
14 return (
15 <div>
16 //other tasks
17 //...
18 <PocketClientTask inDialog={this.props.inDialog} title="A
    sample task" pocket={this.props.pocket}
    plugin={this.props.plugin}
    taskType={ClientTaskType.Sample}
    prediction={this.props.prediction} compute={() =>
    computeTask(this.props.prediction)}
    renderOnComplete={renderTask}/>
19 </div>
20 );
21 }
22 }
```

3.3.2 Server-side plug-ins

Implementing custom server-side plug-ins is more complicated than the client-side ones. Creating a new server-side plug-in consists of the following steps:

1. Design a public API for requests.
2. Add the routes and implement the API in the Flask application.
3. Create a new Docker container (and possibly volume) for the new plug-in.
4. Connect the new container to the existing Docker-compose network.
5. Bind the Celery configuration files to the new container.

6. Implement the wanted functionality in the Docker container.
7. Introduce the new server task to the frontend components and enums.
8. Provide communication between the frontend and the new server-side task via API calls.

Implementing a new server-side plug-in is a complex task and different plug-ins may require different approaches. Therefore, it is highly recommended to have a look at our molecular docking example that is described in detail in section 2.2.2 to understand the integration process.

Conclusion

There were two main goals of this thesis. The first one was to update the existing PrankWeb frontend with newer technologies and to improve the visual design of the application. This was done by introducing the 1D RCSB Saguaro Viewer and Mol* libraries for the structure visualization and by using React to provide a good-looking and responsive user interface. The second goal was to introduce the possibility to add new plug-ins to PrankWeb that allow further postprocessing of the pockets. This was done by creating two interfaces for client-side and server-side plug-ins.

Both of the goals were successfully achieved. The new frontend is more user-friendly and thanks to the used libraries, the visualization is faster than ever before. The plug-in system allows the developers to easily add new features to PrankWeb and deploy these features to their specific users.

Although the current state of PrankWeb is good, there is still room for improvement. There are multiple possibilities to improve the application, on the frontend, the results of docking could be visualized in the Mol* viewer, the 1D viewer could be improved by resizing dynamically according to the window size, and the options for the user could be extended. On the backend, we could add a better program for the docking plug-in, add more plug-ins, and improve the performance of the application.

Still, we believe that the current state of PrankWeb has improved and the application is ready to be used by the scientific community.

Bibliography

- [1] Jianyi Yang, Ambrish Roy, and Yang Zhang. “Protein–ligand binding site recognition using complementary binding-specific substructure comparison and sequence profile alignment”. In: *Bioinformatics* 29.20 (Aug. 2013), pp. 2588–2595. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btt447. eprint: https://academic.oup.com/bioinformatics/article-pdf/29/20/2588/48894365/bioinformatics_29_20_2588.pdf. URL: <https://doi.org/10.1093/bioinformatics/btt447>.
- [2] Radoslav Krivák and David Hoksza. “P2Rank: machine learning based tool for rapid and accurate prediction of ligand binding sites from protein structure”. In: *Journal of cheminformatics* 10 (2018), pp. 1–12.
- [3] Lukas Jendele et al. “PrankWeb: a web server for ligand binding site prediction and visualization”. In: *Nucleic acids research* 47.W1 (2019), W345–W349.
- [4] David Sehnal et al. “Mol* Viewer: modern web app for 3D visualization and analysis of large biomolecular structures”. In: *Nucleic Acids Research* 49.W1 (May 2021), W431–W437. ISSN: 0305-1048. DOI: 10.1093/nar/gkab314. eprint: <https://academic.oup.com/nar/article-pdf/49/W1/W431/38842088/gkab314.pdf>. URL: <https://doi.org/10.1093/nar/gkab314>.
- [5] Joan Segura et al. “RCSB Protein Data Bank 1D tools and services”. In: *Bioinformatics* 36.22-23 (Dec. 2020), pp. 5526–5527. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btaa1012. eprint: <https://academic.oup.com/bioinformatics/article-pdf/36/22-23/5526/36856041/btaa1012.pdf>. URL: <https://doi.org/10.1093/bioinformatics/btaa1012>.
- [6] David L Nelson, Albert L Lehninger, and Michael M Cox. *Lehninger principles of biochemistry*. Macmillan, 2008.
- [7] John Jumper et al. “Highly accurate protein structure prediction with AlphaFold”. In: *Nature* 596.7873 (2021), pp. 583–589.

- [8] Vladimir B Sulimov, Danil C Kutov, and Alexey V Sulimov. “Advances in docking”. In: *Current medicinal chemistry* 26.42 (2019), pp. 7555–7580.
- [9] Frances C Bernstein et al. “The Protein Data Bank: a computer-based archival file for macromolecular structures”. In: *Journal of molecular biology* 112.3 (1977), pp. 535–542.
- [10] Paul D Adams et al. “Announcing mandatory submission of PDBx/mmCIF format files for crystallographic depositions to the Protein Data Bank (PDB)”. In: *Acta Crystallographica Section D: Structural Biology* 75.4 (2019), pp. 451–454.
- [11] Philip E Bourne et al. “[30] Macromolecular crystallographic information file”. In: *Methods in enzymology*. Vol. 277. Elsevier, 1997, pp. 571–590.
- [12] David J Lipman and William R Pearson. “Rapid and sensitive protein similarity searches”. In: *Science* 227.4693 (1985), pp. 1435–1441.
- [13] Andrea Vázquez-Ingelmo, Alicia García-Holgado, and Francisco J García-Peñalvo. “C4 model in a Software Engineering subject to ease the comprehension of UML and the software”. In: *2020 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2020, pp. 919–924.
- [14] Alessia David et al. “The AlphaFold database of protein structures: a biologist’s guide”. In: *Journal of molecular biology* 434.2 (2022), p. 167336.
- [15] Andrei Kouranov et al. “The RCSB PDB information portal for structural genomics”. In: *Nucleic acids research* 34.suppl_1 (2006), pp. D302–D305.
- [16] Warren L DeLano et al. “Pymol: An open-source molecular graphics tool”. In: *CCP4 Newsl. Protein Crystallogr* 40.1 (2002), pp. 82–92.
- [17] Liam J McGuffin et al. “IntFOLD: an integrated web resource for high performance protein structure and function prediction”. In: *Nucleic Acids Research* 47.W1 (May 2019), W408–W413. ISSN: 0305-1048. DOI: 10.1093/nar/gkz322. eprint: <https://academic.oup.com/nar/article-pdf/47/W1/W408/28880002/gkz322.pdf>. URL: <https://doi.org/10.1093/nar/gkz322>.
- [18] Jianyi Yang, Ambrish Roy, and Yang Zhang. “Protein–ligand binding site recognition using complementary binding-specific substructure comparison and sequence profile alignment”. In: *Bioinformatics* 29.20 (2013), pp. 2588–2595.
- [19] Gerard Martínez-Rosell, Toni Giorgino, and Gianni De Fabritiis. “Play-Molecule ProteinPrepare: a web application for protein preparation for molecular dynamics simulations”. In: *Journal of chemical information and modeling* 57.7 (2017), pp. 1511–1516.

- [20] Miha Skalic et al. “PlayMolecule BindScope: large scale CNN-based virtual screening on the web”. In: *Bioinformatics* 35.7 (Aug. 2018), pp. 1237–1238. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bty758. eprint: https://academic.oup.com/bioinformatics/article-pdf/35/7/1237/48968406/bioinformatics_35_7_1237_s2.pdf. URL: <https://doi.org/10.1093/bioinformatics/bty758>.
- [21] J Jiménez et al. “DeepSite: protein-binding site predictor using 3D-convolutional neural networks”. In: *Bioinformatics* 33.19 (May 2017), pp. 3036–3042. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btx350. eprint: https://academic.oup.com/bioinformatics/article-pdf/33/19/3036/49041330/bioinformatics_33_19_3036_s6.pdf. URL: <https://doi.org/10.1093/bioinformatics/btx350>.
- [22] Mayya Sedova, Lukasz Jaroszewski, and Adam Godzik. “Protael: protein data visualization library for the web”. In: *Bioinformatics* 32.4 (Oct. 2015), pp. 602–604. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btv605. eprint: https://academic.oup.com/bioinformatics/article-pdf/32/4/602/49017631/bioinformatics_32_4_602.pdf. URL: <https://doi.org/10.1093/bioinformatics/btv605>.
- [23] David Sehnal et al. “LiteMol suite: interactive web-based visualization of large-scale macromolecular structure data”. In: *Nature Methods* 14.12 (2017), pp. 1121–1122.
- [24] Hai Nguyen, David A Case, and Alexander S Rose. “NGLview—interactive molecular graphics for Jupyter notebooks”. In: *Bioinformatics* 34.7 (2018), pp. 1241–1242.
- [25] Veronika Scheuerová. “Webový plugin pro kombinovanou sekvenčně strukturní analýzu proteinů”. Bachelor’s Thesis. Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra softwarového inženýrství, 2021.
- [26] Cha Zhang and Tsuhan Chen. “Efficient feature extraction for 2D/3D objects in mesh representation”. In: *Proceedings 2001 International Conference on Image Processing (Cat. No. 01CH37205)*. Vol. 3. IEEE. 2001, pp. 935–938.
- [27] Pradeep Anand Ravindranath et al. “AutoDockFR: advances in protein-ligand docking with explicitly specified binding site flexibility”. In: *PLoS computational biology* 11.12 (2015), e1004586.
- [28] Greg Landrum et al. “RDKit: A software suite for cheminformatics, computational chemistry, and predictive modeling”. In: *Greg Landrum* 8 (2013).

- [29] Oleg Trott and Arthur J Olson. “AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading”. In: *Journal of computational chemistry* 31.2 (2010), pp. 455–461.
- [30] David Weininger. “SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules”. In: *Journal of chemical information and computer sciences* 28.1 (1988), pp. 31–36.

Appendix A

Attachments

A.1 Source codes

Source codes of PrankWeb are attached to this thesis in the ZIP archive. The archive contains all of the source codes, not only the work done in this thesis as the work was done in a fork of the same repository as the rest of the PrankWeb application. The directories that were added or intensively modified in this thesis are listed below:

- frontend - contains the source codes of the frontend application
- executor-docking - contains the source codes of the docking plug-in
- web-server - contains the source codes of the Flask application

For the individual changed files, check the GitHub commits. For the current state of the source codes, check the GitHub repository.

A.2 GitHub

The source codes are publicly available on GitHub at <https://github.com/cusbg/prankweb>.

The official documentation of the PrankWeb architecture and P2Rank tool is available at <https://github.com/cusbg/p2rank-framework/>.

The development was done in a fork of the PrankWeb repository at <https://github.com/luk27official/prankweb>. The fork branches are continuously merged into the upstream repository after the pull request is approved.

A.3 Abbreviations

- **1D** - one dimensional
- **3D** - three dimensional
- **ADFR** - AutoDockFR
- **API** - Application Programming Interface
- **CIF / mmCIF** - Macromolecular Crystallographic Information File
- **CSS** - Cascading Style Sheets
- **CSV** - Comma Separated Values
- **FASTA** - FAST-All
- **JRE** - Java Runtime Environment
- **JSX** - JavaScript Syntax Extension
- **JSON** - JavaScript Object Notation
- **PDB** - Protein Data Bank
- **PDBx** - Protein Data Bank eXtended
- **pLDDT** - per-residue Local Distance Difference Test
- **RCSB** - Research Collaboratory for Structural Bioinformatics
- **REST** - Representational State Transfer
- **SCSS** - Sassy CSS
- **SMILES** - Simplified Molecular Input Line Entry System
- **WSGI** - Web Server Gateway Interface

A.4 Pocket detail designs

This section contains all pocket detail designs that were considered for displaying more information about a pocket. In the end, the figure A.4 was chosen. For more information refer to section 2.2.

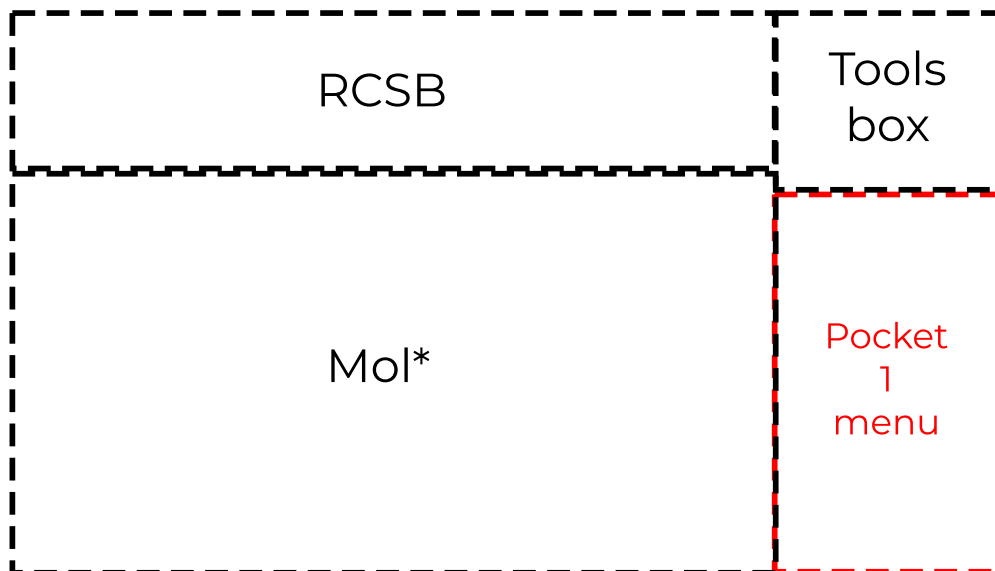


Figure A.1 The first option for displaying pocket information.

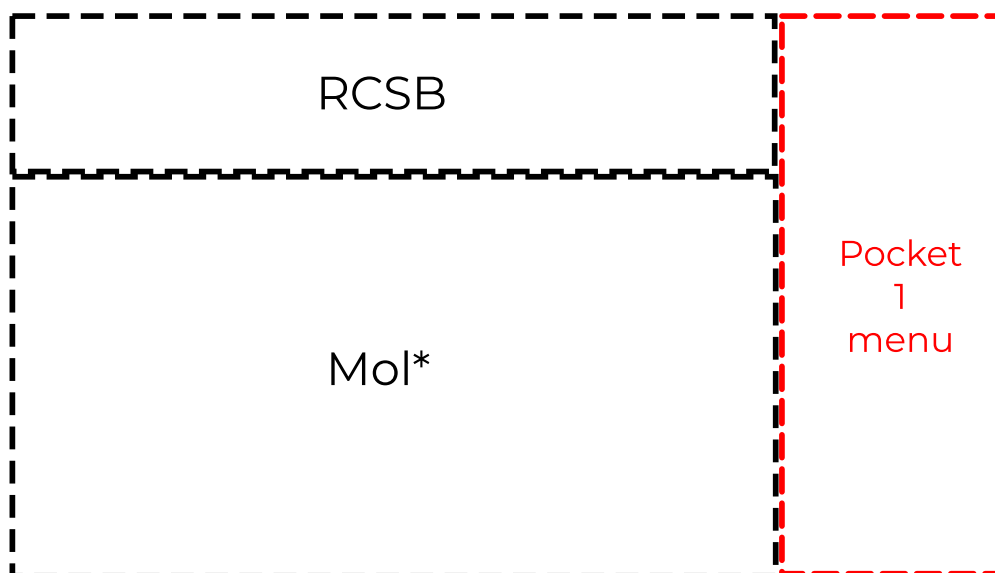


Figure A.2 The second option for displaying pocket information.

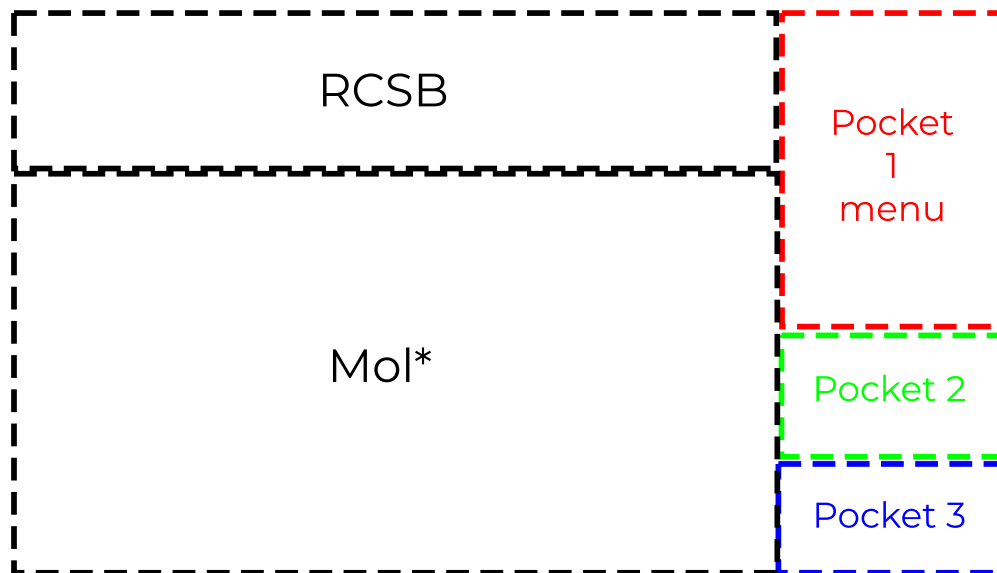


Figure A.3 The third option for displaying pocket information.

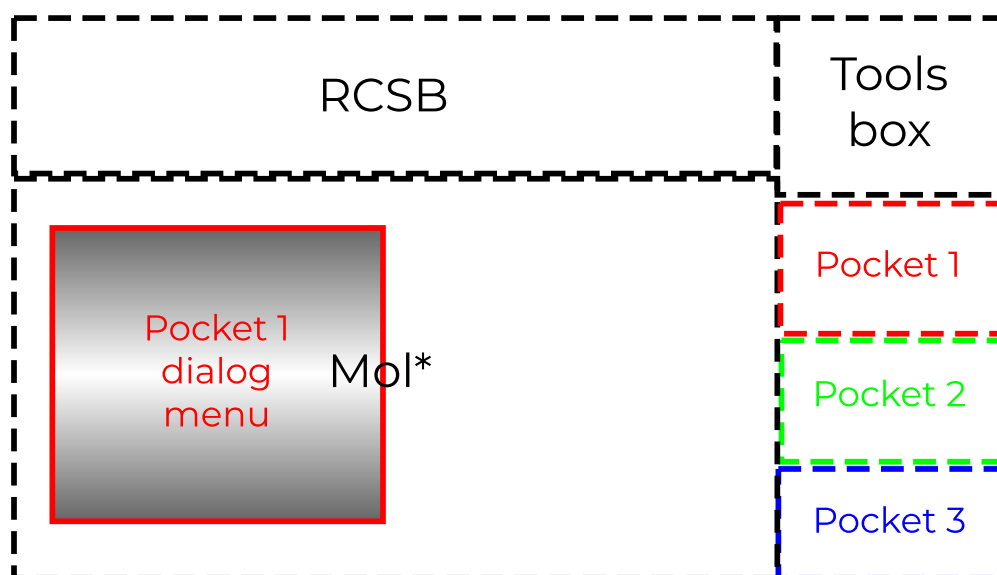


Figure A.4 The fourth and chosen option for displaying pocket information.