Implementation of a Python library for offline utilisation of the VirtualChemistry webserver

Yasmine Odelbrink
Abstract

This thesis discusses the implementation of a Python library, developed to provide an offline alternative to the VirtualChemistry.org website. The Python library interacts with a REST API that allows access to the Alexandria database used in VirtualChemistry.org. The REST API includes various endpoints built using the Express framework in Node.js. The implementation of the Python library involved two main steps: documenting the API and writing the library. The API documentation process included identifying all the endpoints, determining their parameters and response formats, and writing documentation for each individual endpoint. The Python library was implemented by establishing a connection between the library and the API server using the requests library. The library class, named Molecular, encapsulates various methods representing different API endpoints. The architectural design of the library follows a client-server model, where the Python library acts as the client interacting with the server-side. The library’s functionality was assessed by comparing it to the website’s functionality, highlighting the differences and similarities between them. The library was evaluated in terms of its functionality, error handling, performance and documentation. Overall, the implemented Python library successfully interacts with the API for the molecular database, providing users with a convenient and efficient offline alternative to using the VirtualChemistry.org website.
4.2.3 Architectural Design ........................................... 19

5 Evaluation of the Python Library ................................ 25
  5.1 Functionality ................................................. 25
  5.2 Error handling .............................................. 28
  5.3 Performance ................................................ 28
  5.4 Documentation .............................................. 30

6 Further work ..................................................... 31
  6.1 Error handling .............................................. 31
  6.2 Secure connection .......................................... 31

7 Conclusion ......................................................... 32

References .......................................................... 33
  Alexandria ........................................................ 35
  Python ............................................................ 35
  TypeScript ......................................................... 36
  Background ....................................................... 36
  Evaluation ......................................................... 37
  The Website ....................................................... 37
Chapter 1

Introduction

1.1 Purpose and Goal

The purpose of this thesis is to develop a Python library that enables offline access to the services provided by Virtualchemistry.org\[25\]. The thesis seeks to leverage Python’s popularity, flexibility, and its cross-platform compatibility to create a flexible and user-friendly library that integrates well with the existing workflow.

1.2 Problem Description

The website called Virtualchemistry.org is a service targeted towards computational chemists, and also, for example, the pharmaceutical chemistry community. The website provides chemical data for a large number of molecules alongside data files to perform numerical simulations on those molecules. The website is hosted at Uppsala University by the research group of David van der Spoel at the Department of Cell and Molecular Biology. A screenshot of the homepage of the website is shown in fig. 1.1.

For a previous project started two years ago, the code behind the webserver was modernised using the client-server architecture REST API. This update aimed to enhance the website’s efficiency and improve the user experience.

While the new and improved version of the website is still in its prototype stage and has not fully replaced the old version, it offers a wide range of functionalities
such as retrieving information from the Alexandria database and performing calculations. The new website can only be accessed through a local connection. The homepage of the newer version is shown in fig. 1.2.

Once the new website has been completed and deployed online, all these functionalities will be accessible over the internet. However, there is also a desire to provide offline access to the same set of functionalities. To facilitate offline accessibility, it was decided that a Python library should be developed. The aim of this thesis is to develop and implement such a library.

In the following chapters, any references made to the website and its functionality pertain specifically to the new and enhanced prototype version.
VirtualChemistry Home Alexandria News Compound Database Properties Studies Force Field Packages Correlation Tool Interactive F.A.Q. More »

Home

Virtual chemistry here refers to molecules including both covalently bound compounds and ionic systems in a computer rather than in a laboratory. We try to predict properties of compounds using theoretical models that are calculated or simulated in a computer.

The website features:
- Results from classical molecular dynamics simulations of liquids with experimental data to compare results to.
- QNACCS input files for simulations corresponding to the results. Currently three force fields are supported: OPLS, QM7, and DIPPO. We provide validated input files in order to make sure that results based on these files are consistent with the force field used.
- Results from gas phase quantum chemistry calculations using a range of methods. At a later stage we will provide input files for reproducing these calculations as well.
- A tool to make correlation plots between the available data sets.

Experimental as well as theoretical scientists can use this resource in order to check how well a certain molecular species/compound is reproduced in a specific force field or quantum chemistry calculation. Since we provide experimental and calculated or simulated physical properties side-by-side, you can form your own opinion. Obviously, these numbers can be used for force field benchmarking and development using other software packages as well.

Enjoy!

Database Statistics
Compounds with predicted properties in the database: 4600

![Database Statistics Chart]

Figure 1.2: Home page of the updated and enhanced version of the Virtualchemistry.org website.
1.3 Methodology

The development of the project will be done in the following steps. First, by analyzing and learning the website’s functionality, this includes documenting the existing API. The next step is the implementation of the library. And finally, the efficiency of the implemented library will be evaluated in Chapter [5]. The aspects which will be considered are these four aspects:

**Functionality:** It is important to evaluate if the library provides the expected functionality and verify if it meets the requirements of the intended use. Thus this is probably the most important aspect in evaluating the library. The evaluation can be done by reviewing the implemented methods and their corresponding functionality, and comparing them to the functionality of the website.

**Error Handling:** This aspect assesses how the library handles errors and exceptions.

**Performance:** The performance of the library can be evaluated by measuring the response times for various requests and assessing whether they meet the desired performance criteria.

**Documentation:** Documentation is essential to any coding project, and the quality and clarity of the documentation should be evaluated to ensure that users can easily understand how to utilise the library.

In addition to these four aspects, it is worth mentioning two other aspects in evaluating the library, that due to the time constraint in this thesis will not be considered here:

**Stability and Reliability:** The stability and reliability of the library can be assessed by using real-world scenarios and monitoring the functionality over time. This can help identify any potential bugs, inconsistencies, or issues that may arise after a while. This aspect will however not be evaluated or discussed further in this essay because the library has not yet been released for utilisation, hence the stability and reliability aspect cannot be properly evaluated.

**Usability:** The library should be user-friendly. Evaluating the library’s ease of use, readability, and overall developer experience can help determine its usability. The reason this aspect won’t be evaluated is because of time-constraint in having users try and test the library.
Chapter 2

Background

This chapter covers the existing webserver VirtualChemistry.org (2.1), describes what REST APIs are (2.2), and explains Python libraries and the reasons for choosing Python (2.3). The last part (1.3) describes the methodology that will be utilised in Chapter 5 to discuss and evaluate the resulting library.

2.1 VirtualChemistry.org

The reason for creating the VirtualChemistry.org website was so that computational chemists and the pharmaceutical chemistry community could utilise and access reliable and accurate data through the Alexandria database (further discussed in 2.1.1) efficiently. By providing such data sets, researchers can simulate chemical reactions and predict molecular behavior with greater precision, efficiency and reproducibility.

2.1.1 Alexandria

The Alexandria library[1] is a publicly available quantum-chemical database which is at the core of the website, providing necessary data for retrieving and displaying information. It contains a wide range of data about molecular properties such as optimised shapes, frequencies, thermochemistry, polarizability, and electric moments. The library is being continuously updated by the team that works on it. It is built using SQL and managed by the relational database management system SQLite.
2.2 REST APIs

A REST API[11] is a way for two software applications to communicate with each other. The term "REST" stands for Representational State Transfer, and is a design principle for creating web services. The primary goal of a REST API is to provide a standardized way for applications to interact with each other, regardless of the programming languages or platforms they use.

A REST API works by using the HTTP protocol to make requests and receive responses. When one application wants to communicate with another application, it sends an HTTP request to the other application’s REST API endpoint, i.e. the URL with the location where the API interacts with the client. This request contains information about what the first application wants to do, such as retrieving data, or performing an action.

The REST API on the receiving end then processes the request and sends back an HTTP response. This response contains the requested data or information about whether the requested action was successful. The response is usually in a format that both applications can understand, such as JSON or XML.[7]

2.3 Python

Python is chosen for this project since there are several advantages when using it to create a library. Python is widely recognized as one of the most popular programming languages today[24]. This popularity translates into a strong community support system, ensuring that users will have access to plenty of resources, tutorials, and forums. If users encounter issues or have questions regarding the usage of the library, they can find help from the community.

Secondly, Python is a good choice for consuming APIs [18], since it comes with plenty of built-in packages to use. One of these packages is the ’requests’ library, which can be used to send requests to APIs [8]. Using a programming language with such a popular and well-kept built-in package is quite useful. Apart from the ’requests’ library, Python has many other libraries and frameworks that can be utilised. Python offers integration with numerical and scientific computing libraries such as NumPy, SciPy, and Pandas. By utilising these libraries alongside the Python library, users can perform complex calculations on the chemical data provided by the database. Libraries are discussed in more detail in [23.1]
Thirdly, Python is commonly regarded as, and aims to be, a simple and readable programming language, and its simplicity and readability allow both experienced developers and beginners to quickly grasp and utilise the library.

Lastly Python’s cross-platform compatibility ensures that the library can be used on different operating systems such as Windows, macOS, and Linux without major modifications. This allows researchers to access the library offline regardless of their preferred operating system. [10]

2.3.1 Python Libraries

In the context of Python, a library is a collection of related modules that provide useful functionality, such as data processing, visualization, machine learning, or web development. [5]

Python libraries are distributed through various channels, such as the Python Package Index (PyPI), GitHub, or other online repositories. [15] [17] Installing a library is usually a straightforward process, and can be done using a package manager such as pip, conda, or Anaconda. [6]

Once a library is installed, it can be imported into a Python script or application, and its functions and classes can be used to accomplish various tasks. Libraries often have extensive documentation and examples that make it simple for users to get started and learn how to use them.

Python libraries can be very powerful and can save developers a lot of time and effort, as they can provide pre-built solutions to common problems. Using a library can also help ensure that code is written in a more efficient and reliable way, as libraries have often been tested and optimized for performance. [26]

2.3.2 requests Library

The requests library in Python is a commonly used HTTP library for making HTTP requests to web servers. It simplifies the process of sending HTTP requests and receiving responses by abstracting away many of the low-level details. The requests library allows users to send GET, POST, PUT, DELETE, and other HTTP methods with different parameters such as headers, cookies, and authentication. It also provides support for handling various types of data including JSON, form encoded data, and binary files. [19]
Chapter 3

Existing Methods Used in VirtualChemistry.org Website

This chapter describes the implementation and functionality of the already existing website, VirtualChemistry.org. The chapter covers how the prototype of the website is run, the functionalities that it provides [3.1] the existing API [3.2] and the plan to document the API.

3.1 The webserver

3.1.1 How to run the webserver

Before proceeding with the implementation of the Python library, it is crucial to establish a connection to the server so that the Python script can access the API endpoints.

Since the new website has not yet been deployed, the only way to access it is by running it on a local server. It is run in the development environment, or 'dev environment', which is sufficient for accessing the website locally and utilising most of its functionality. A dev environment is a setup that developers use to create and test software. It provides a controlled space where developers can work on the code, make changes, and test it without affecting the live environment[27].

The following are prerequisites to run the dev environment for the webserver: Node.js, npm package manager, Python 2, and zip.
To use the website, port 4200 and 8008 need to be available. During the implementation and creation of the library, the website was utilised to see its functionalities as well as to compare them to the functionality of the resulting library. However, if one only wishes to utilise the now implemented Python library, it is not necessary to host the website in the background, only the API needs to be hosted on port 8008. Therefore, the only steps needed to be followed for utilising the Python library are steps 1-4.

**How to run the dev environment:**

To run the dev environment, the following steps are used:

1. Place the 'Alexandria.dat' (Alexandria database) file in the vchemistry-back directory
2. Open terminal in the vchemistry-back directory.
3. Install all prerequisites: npm install
4. Run command: npm run dev
5. Open another terminal in vchemistry-front directory.
6. Install all prerequisites: npm install
7. Run command: npm run dev

This will make the backend run on port 8008, and the front end running on port 4200. The website can be found on http://localhost:4200/.

There is also a possibility to run something called the ‘production’ environment. This is however not necessary for developing or utilising the Python library, and hence won’t be further explained in this paper.

### 3.1.2 Functionality of the website

The website currently has a lot of functionality that can be used together with the Alexandria database. When opening the website, the server directs you to the home page with url http://localhost:4200/home. Right on the home page, three diagrams can be seen. These diagrams showcase molecule statistics from information taken from the Alexandria database, shown in fig. 3.1.
Figure 3.1: Database Statistics of the Alexandria database as shown on the 'Home' page of the website

Under the title, there is a text saying "Compounds with predicted properties in the database: 4600" The number 4600 is the total number of molecules with properties in the database, and this statistic is rendered calling the API endpoint /molecule_count. As mentioned, the database is being continuously worked on, which means that the number of molecules in the database may not always be 4600. Given that the website calls the API endpoint each time the page is loaded, the number of molecules is a dynamic number which changes as the database changes. This is the case for all functionality on the website which utilises the API. The diagrams with statistical info are rendered by calling the API endpoint /phasestats/:id.

On the 'Compound Database' page, with url 'http://localhost:4200/moldb', the user can search through and see all the molecules in the database (fig. 3.2). There is also functionality that allows the user to search for specific molecules. These molecules are rendered by calling the API endpoint /moldb/main.

Additionally on that same page, the user can choose to only see molecules in either their liquid or gas form, or all the molecules. Even more, the user can choose to filter the molecules through various more parameters under the tab 'More filters', e.g. to only show organic or inorganic molecules, or both. The user can also choose to filter the molecules through their class. (fig. 3.3). This filtering is rendered by calling the API endpoint /moldb/filter.
Figure 3.2: List of all molecules in the Alexandria database as shown on the 'Compound Database' page of the website

Figure 3.3: Filters for the molecules on the website
3.2 The API

This section covers the existing API in more depth, how it is built and some of the functionality that it provides.

3.2.1 TypesScript

The API is written in TypeScript, which is a programming language that extends JavaScript with static typing (meaning it shares the same base syntax as JavaScript, but adds static typing). It transpiles to JavaScript [21] for compatibility with web browsers. [22] When a language is transpiled, it means that one higher-level language is converted into another higher level language. [16] In this case, TypeScript is converted into JavaScript. [2]

3.2.2 Endpoints

The API includes various endpoints that allow interaction with the system and retrieval of data. The API is built using the Express framework, a popular choice for building web applications in Node.js. [14] Listing 3.1 provides a glimpse into the implementation of the API’s endpoints, showing the first couple lines of code in the API.
import express from 'express';
import Database from 'better-sqlite3';
import { Mutex, Semaphore } from 'async-mutex';
import * as fs from 'fs';

// ...

/** Router to respond to API calls */
const router = express.Router();

const dbOptions = {
  verbose: console.log,
  fileMustExist: true,
};
/** Alexandria database */
const db = new Database('AlexandriaDB.dat', dbOptions);

// Homepage statistics (molecule count)
router.get('/molecule_count', (req, res) => {
  // Query to retrieve the count of molecules in the 'molecules' table
  const sql: string = "select count(*) as total from molecules";
  const row = db.prepare(sql).get();
  // Respond with the count as JSON
  res.json({
    "message": "success",
    "data": row
  });
});

// ...

Listing 3.1: First few lines of code in the API.

To explain the code snippet a bit; it begins by importing the necessary modules and libraries. The express module is imported from the 'express' package, the Database class is imported from the 'better-sqlite3' package, the Mutex and Semaphore classes are imported from the 'async-mutex' package, and the whole fs module is imported. Then, an instance of the Express router is created using express.Router(), which handles incoming API requests. After this, dbOptions is defined, which is an object that specifies options for the database connection.

Then, a new instance of the Database class is created, representing the Alexandria
database. It takes two parameters: the name of the database file (AlexandriaDB.dat) and the dbOptions object. This establishes the database connection to 'AlexandriaDB.dat' using the 'better-sqlite3' library.

Lastly, the code defines a route for handling a GET request to the endpoint /molecule_count. When a request is made to this endpoint, the code executes a SQL query to retrieve the count of molecules from the 'molecules' table in the database. The result is then sent as a JSON response containing a success message and the data (the count) in the response body.

These initial lines of code set the foundation for implementing other endpoints in the API. Each endpoint can have its own route and logic to handle specific requests and provide relevant data or perform certain actions, with a structure similar to the structure of the /molecule_count endpoint.

### 3.2.3 Structure of the API endpoints

The endpoints in the API have a structured format that follows the Express framework. Each endpoint is associated with a specific route and HTTP method (e.g., GET). The endpoint handler function is responsible for performing the necessary operations, such as querying the database or reading files, and returning a response in JSON format. The response typically includes a success message and the requested data. Here is an overview of how the structure of an endpoints handler function looks like:

1. Route definition, shown in listing 3.2, which is where the router handles different HTTP methods such as GET, POST, etc. In the listing, the endpoint is defined using the get method, specifying the route path ("/endpoint").

   ```javascript
   router.get("/endpoint", (req, res) => {
     // Endpoint logic goes here
   });
   ```

   Listing 3.2: Route definition.

2. Endpoint Logic: Inside the route handler function, the necessary logic is implemented to handle the request and generate a response. This typically involves interacting with the database or performing other computations. The majority of the functions interact with the database, hence this endpoint logic often involves SQL queries to fetch data from the database. The queries are constructed using template literals to dynamically insert values.
The results are retrieved using the all or get methods and returned as a JSON response. An example is shown in listing 3.3.

```javascript
const sql = SELECT ... FROM ... WHERE ...;
const row = db.prepare(sql).all(req.params.id);
```

Listing 3.3: Database operations.

3. The response is sent back to the client in JSON format, typically containing a "message" field indicating the status of the request (e.g., success or error) and a "data" field containing the relevant data. A typical response is shown in listing 3.4.

```javascript
res.json({
    "message": "success",
    "data": row
});
```

Listing 3.4: Example of the response.

In total, the API is written in 622 lines of code, hence why all endpoints and their functionality are not displayed and explained. In total there are 21 endpoints and one async function in the API. 20 of these endpoints make GET requests, and one of them makes a POST request. The URLs of the endpoints are as follows:

- /molecule_count
- /phasestats/:id
- /moldb/main
- /moldb/classes
- /moldb/filter
- /moldb/:id
- /molecule/summary/:filename
- /molecule/3d/:filename
- /molecule/synonym/:molid
- /mol/table
- /mol/tablerow
- /datasource
- /plotpoints
- /mol/proptypes/:phasename
- /ref/:reference:
- /refs
- /refs/doi
- /properties/:phasename
- /newactrunid
- /actrun-download/:id
- /actrun-upload/
Chapter 4

Implementation of the Python library

This section covers how the work of this thesis, the Python library, was implemented and distributed. The implementation of the Python library to add functionality for offline use is covered in the following two steps:

1. Documenting the API (4.1)
2. Writing the Python library (4.2)

4.1 Documenting the API

The Python library needs to communicate with the API endpoints because all of the methods in the library will be implemented dynamically. That is, the methods will retrieve information form the database each time they are called. The only way to access the database using a Python script is by connecting the Python script to the API. Therefore, when developing a Python script that utilizes API endpoints, it is essential to have a clear understanding of these endpoints. This includes knowing what they return and what the return type is, as well as any required or optional parameters that it needs.

Consequently, the first step to creating the Python library is to understand the API and its endpoints, so that the methods in the library can be properly implemented. There was no previous documentation of the API and its endpoints, thus the first
The first step to creating the library was to document the API and, most importantly, to document its endpoints. To create the documentation, the following steps were taken:

1. Identify the endpoints: Make a list of all the endpoints and their corresponding HTTP methods (e.g. GET, POST, PUT, DELETE).

2. For each endpoint, determine what parameters are required, what parameters are optional, and what format the response will be in. Make a note of all the possible parameter values and the expected response formats.

3. Write the documentation: Once you have identified all the endpoints, parameters, and response formats, start writing the documentation.

4. Test the documentation: After writing the documentation, it’s important to test it by trying out the API using the documented examples. This will help identify any mistakes or omissions in the documentation and ensure that it accurately reflects the API’s behavior.

The resulting documentation in this work was created by following these steps for all the individual endpoints. The majority of the functions utilise SQL commands to get information from the database, and for these functions the SQL commands were carefully studied to figure out what the input and the response would look like. By doing this, all the endpoints were properly documented. To help visualize the documentation for the endpoints, an example is shown in listing 4.1.

```plaintext
# Endpoint: /molecule_count
* Description:
Returns the total number of molecules stored in the database.
* HTTP Method:
GET
* Response:
message: A string value indicating the status of the API call. If the API call was successful, the value of this property will be "success".
data: An object containing the total number of molecules stored in the database.
* Parameters
None
```

Listing 4.1: API documentation example
4.2 Writing the Python library

Once the API documentation exists, the development of the Python library can begin. The first step was to get the server up and running (section 4.2.1) so that the Python script could establish a connection with the API.

4.2.1 Starting the server

To develop a working Python library, there needs to be a connection between the library and the API server, since the methods in the library need to utilise the API endpoints.

The first step to establish a connection is to ensure that TypeScript is installed. This can be done using the following command:

\[ \text{tsc --version} \]

which will display the version of TypeScript installed, if it is installed. If TypeScript is not installed, an error message will be displayed, indicating that the command "tsc" is not recognized. Alternatively, the command npm list can also be used to check if TypeScript is installed locally within the project. When the command:

\[ \text{npm list TypeScript} \]

is run from the root directory it will show the installed version of TypeScript and if it is present in the project’s dependencies. If TypeScript is not installed, there will be a message indicating that the package is not found.

If TypeScript is not installed, it can be installed using these steps:

1. Install Node.js (If Node.js is already installed, this step can be skipped): To install Node.js, visit the official Node.js website [https://nodejs.org](https://nodejs.org) and download the appropriate installer.

2. Verify Node.js installation: After installing Node.js, open a command prompt or terminal and run the following command to verify that Node.js is installed correctly:

\[ \text{node -v} \]

3. Install TypeScript globally: Once Node.js is installed, TypeScript can be installed globally by running the following command:
npm install -g TypeScript

Now that TypeScript is installed, the backend server can be started by following steps 1-4 in 3.1.1.

If the server is running properly, the API server side should be hosted on localhost:8008. This is necessary to establish the connection, so it is important to make sure that it is running.

4.2.2 How the library was implemented

Once the API was documented, and the API server was running, the implementation of the methods started. First of all, the connection to the API from the Python script was done using the requests library. Connecting to the API using the requests library is quite straightforward. Depending on the method (GET or POST) either the function requests.get(url, params=params) or requests.post(url, params=params) will be utilised. The two functions require the API’s URL as a required argument, while the parameters can be provided as optional arguments. These functions are the only ones needed to establish a connection with the API.

With this in mind, the remaining implementation steps were fairly straightforward. By referring to the API endpoint documentation the required parameters and expected response could be deduced. Knowing the parameters and expected response, the methods could be created accordingly, and testing was conducted to ensure their proper functionality and accurate return of results. Further details on the architectural design of the library can be found in 4.2.3.

4.2.3 Architectural Design

This section covers the overall architecture and structure of the implemented Python library. The structure for the library can be summarized as follows:

Class Structure: The library is implemented as a class named Molecular with various methods representing different API endpoints for interacting with the molecular database. To use the library, an instance of the class needs to be created, as shown in listing 4.2. Once the instance is created, all the methods can be applied to 'client' and used.
client = Molecular("http://localhost:8008/api")

Listing 4.2: Example of initiating an instance of Molecular class

**Initialization**: The class constructor `__init__` takes a `base_url` parameter, which represents the base URL of the API. The method is shown in listing 4.3

```python
def __init__(self, base_url):
    self.base_url = base_url
```

Listing 4.3: Init method

**Request Methods**: The library provides two methods for making HTTP requests: `make_get_request` and `make_post_request`. One handles GET requests, while the other handles POST requests. These methods handle the low-level details of making requests using the `requests` library, such as constructing the URL, sending the request, and handling the response. The responses are checked for success by examining the status codes. If a successful response is received (status code 200), the data is extracted from the JSON response and returned. See fig. 4.1 and fig. 4.2 for a visual description of the implementations of these methods.

**Methods**: The library includes a set of methods that correspond to different API endpoints. These methods encapsulate the logic for constructing the specific endpoint URLs and calling the request methods to interact with the endpoint logic in the API. If not counting the requests methods, there are 22 methods in total. Examples of these methods include `get_total_molecule_count`, `get_phasetstats`, `get_all_molecules_with_properties`, and many more. They utilise the `make_request` and `make_post_request` methods to make the corresponding HTTP requests and return the response data. If an error occurs during the request or response processing, `None` is returned.

**Endpoint Parameters**: Some methods either require, or provide optional parameters that are passed as query parameters in the API requests. These parameters are incorporated into the request URL using the `params` argument of the request methods.

**Additional Functionality**: The library has only two methods which make POST requests to the server, these are the `input_smiles` and `upload_file` methods. The `input_smiles` sends in a SMILES (Simplified Molecular Input Line Entry System) string, which is a chemical notation that allows a user to represent a
Figure 4.1: Flowchart of the make_get_request method
Figure 4.2: Flowchart of the make_post_request method
chemical structure in a way that can be used by the computer. The method then returns a JSON dictionary with several values, mostly chemical and molecular information.

The `upload_file` method uploads files to a selected run. The files get uploaded to the directory `runs`. The method also returns the filepath, as well as the content of the file.

For a better visual representation of these methods, see fig. 4.3 which is a flowchart which combines the two methods.

Overall, the architecture of the library follows a client-server architecture where the Python library acts as the client that interacts with the server-side API. The `Molecular` class serves as an interface for making requests to various endpoints of the molecular database API. To use the library, an instance of the `Molecular` class is created with the base URL of the remote server. Then, the desired methods can be called to retrieve specific molecular data or perform actions such as downloading ACT runs or uploading files.

The library utilises the `requests` module to send HTTP requests to the server. The base URL which is provided during initialization is used as the prefix for constructing the complete URL for each API endpoint.
Figure 4.3: Flowchart of the upload_file and input_smiles method
Chapter 5

Evaluation of the Python Library

This chapter aims to evaluate and discuss the effectiveness and performance of the library. In section 1.3, it was already mentioned which aspects will be considered when evaluating the library. In the sections below, these aspects will be applied to the library and discussed.

5.1 Functionality

To evaluate the functionality, comparisons will be drawn between the website’s functionality, and the library’s. One notable distinction between the two is the absence of visual representation of the API responses in the library. For example, one of the functionalities that the website offers is the possibility to plot chemical data in a correlational tool, as depicted in fig. 5.1. Although implementing this feature in Python is feasible, it was not desired. As an alternative, the library offers a corresponding function called arguments:get_plot_data, which accepts the following seven arguments:

- \texttt{phaseidX}: The ID of the phase for the X-axis property.
- \texttt{phaseidY}: The ID of the phase for the Y-axis property.
- \texttt{propidX}: The ID of the X-axis property.
- \texttt{propidY}: The ID of the Y-axis property.
- \texttt{srcidX}: The ID of the data source for the X-axis property.
• srcidY: The ID of the data source for the Y-axis property.

• classname: The name of the class for highlighting points.

In a sample run, only a portion of the response is displayed in fig. 5.2 due to the extensive length of the complete response. As can be seen, the response appears quite dense and not as easily comprehensible as a visual plot, like the one on the website. However, if a visual plot is desired, it can be achieved by creating a plotting function which utilises the get_plot_data, and uses the response to generate a graph in Python.

A part from the visual aspect, the library provides all the functionality that the website offers. While the API has 21 endpoints, the library includes 22 methods. The reason for this is that the methods upload_file and input_smiles both utilise the /actrun-upload endpoint. In theory, these two methods could be combined into a single method. For better usability the methods were created as separate methods. Consequently, all API endpoints are utilised within the library.
Figure 5.1: Screenshot of the ‘Correlational Tool’ from the website
5.2 Error handling

The most developed error handling is in the make_request and make_request methods. The error handling for make_request shown in listing 5.1 and the error handling in make_request is identical except for the second line which reads:

response = requests.post(url, data=payload, headers=headers, params=params).

```python
try:
    response = requests.get(url, params=params)
    response.raise_for_status()
    data = response.json()
    return data["data"]
except requests.exceptions.RequestException as e:
    print("An error occurred during the request:", e)
    return None
except (KeyError, ValueError) as e:
    print("An error occurred while processing the response:", e)
    return None
```

Listing 5.1: Error handling in make_request function

The error-handling includes try-except blocks, requests.exceptions.RequestException as well as KeyError and ValueError exceptions and appropriate error messages.

5.3 Performance

The make_get_request and make_post_request methods are responsible for making API requests, and their response times can be measured using appropriate tools or by adding timing logic to the code. The most commonly used ways of measuring the performance of a function or method in Python are:

28
• Using the `time.perf_counter()` function.
• Using the `timeit` module.
• Using the `profile` and `cProfile` modules.

In this section, the `timeit` module will be utilised. The module was tested on the two methods (`upload_file` and `input_smiles`) which both utilise the `make_post_request`. This was done to test if the response time is the same. If the response time is similar, it would indicate that the performance is mostly dependent on the efficiency of the 'requests' methods and the performance can then be used to evaluate the library. However, if the response times differs, it would indicate that the performance is mostly affected by the code in the API, and hence not possible to change by adjusting the 'request' methods.

Each method was tested 10 times using the `timeit` module, and an average of these times was calculated. The results are displayed in table 5.3a and 5.3b.

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.071</td>
</tr>
<tr>
<td>2</td>
<td>0.085</td>
</tr>
<tr>
<td>3</td>
<td>0.091</td>
</tr>
<tr>
<td>4</td>
<td>0.054</td>
</tr>
<tr>
<td>5</td>
<td>0.093</td>
</tr>
<tr>
<td>6</td>
<td>0.058</td>
</tr>
<tr>
<td>7</td>
<td>0.096</td>
</tr>
<tr>
<td>8</td>
<td>0.051</td>
</tr>
<tr>
<td>9</td>
<td>0.072</td>
</tr>
<tr>
<td>10</td>
<td>0.065</td>
</tr>
<tr>
<td>Average</td>
<td>0.074</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Execution Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.967</td>
</tr>
<tr>
<td>2</td>
<td>0.326</td>
</tr>
<tr>
<td>3</td>
<td>0.997</td>
</tr>
<tr>
<td>4</td>
<td>0.987</td>
</tr>
<tr>
<td>5</td>
<td>1.042</td>
</tr>
<tr>
<td>6</td>
<td>0.984</td>
</tr>
<tr>
<td>7</td>
<td>1.039</td>
</tr>
<tr>
<td>8</td>
<td>0.394</td>
</tr>
<tr>
<td>9</td>
<td>0.978</td>
</tr>
<tr>
<td>10</td>
<td>0.333</td>
</tr>
<tr>
<td>Average</td>
<td>0.805</td>
</tr>
</tbody>
</table>

(a) Execution times when calling `upload_file`  
(b) Execution times when calling `input_smiles`

Figure 5.3: Comparison of execution times

The code that makes up the two functions are similar; they both take in an argument and send this argument to the API endpoint `/actrun-upload`. The function `upload_file` has two additional lines of code which read the file that was given. Inside the endpoint `/actrun-upload`, there are `if` statements which specify how to proceed based on whether the given argument is a file or a SMILES string.
If the response from the endpoint was independent on the argument, the execution times of the functions upload_file and input_smiles should be similar. However, as evident by comparing the two tables, the performance of the two methods differs by quite a lot. The input_smiles method is more than 10 times slower than the upload_file method, despite these methods both calling the make_post_request method. This indicates that the performance is more dependent on what happens in the API, hence these results show that evaluating the performance of the library is not very useful because the API slows down the response significantly either way.

5.4 Documentation

The documentation should be comprehensive and structured for every method. Therefore, the documentation follows a consistent structure for every method. It provides plenty of information; what the method does, required or optional arguments, expected return values, and examples for better comprehension. Fig. 5.2 showcases the documentation for the first method get_total_molecule_count. All documentation in the other methods are implemented in exactly the same way.

This approach arguably ensures that the code is well-documented, as it provides the users with extensive documentation for every available method. The consistency of the documentation allows users to understand it once and apply that understanding to all the methods.

```
""
    Returns the total number of molecules in the database
    Returns:
    The total number of molecules or None if the request was unsuccessful.
    Return type:
    int or None
    Examples:
    >>> api = Molecular("http://localhost:8008/api")
    >>> api.get_total_molecule_count()
    {'total': 4600}
    ""
```

Listing 5.2: Documentation for the get_total_molecule_count method
Chapter 6

Further work

This section discusses what improvements that could be made to the library, and further work that might be considered for the future.

6.1 Error handling

The library does contain error handling, as discussed in 5. Further work could include testing more cases to find potential errors. The errors might come up naturally during usage of the library, but more test cases could also be created as a preventative measure.

6.2 Secure connection

Further work could be done in ensuring the connection between the library and the API is secure. In this work there was no evaluation of the security of the library due to time constraints, and it is unclear whether it might be possible for an unauthorized person to gain access to sensitive information transferred between the library and the API. This might be looked over and analysed.
Chapter 7

Conclusion

The goal of this thesis was to develop a Python library that would provide the same functionality as the VirtualChemistry.org website. The development process consisted of two steps; firstly, the API endpoints were documented, and this documentation served as a foundation for the subsequent implementation of the library. Secondly, the library itself was written. The resulting library consists of 22 methods, exceeding the 21 methods available in the API. This shows that the library utilises all of the API endpoints, and even implemented two methods for the same endpoint. With the exception of two methods, each method in the library corresponds to a unique endpoint in the API. This demonstrates that the library effectively utilizes the API’s full range of capabilities, allowing it to mirror the functionality on the VirtualChemistry.org website. When evaluating the library’s functionality, it was noted that some visual features available on the website, such as plotting chemical data, are not included in the library. However, the library offers equivalent functions that allow the users to plot the data from the response if they wish to do so.
References


Alexandria


Python


TypeScript


Background

Evaluation


The Website