MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Web Application for Collecting and Unifying Photometric Data from Multiple Astronomical Archives

Bachelor's Thesis

ONDŘEJ MAREK

Brno, Fall 2025

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Web Application for Collecting and Unifying Photometric Data from Multiple Astronomical Archives

Bachelor's Thesis

ONDŘEJ MAREK

Advisor: RNDr. Martin Kuba, Ph.D.

Department of Computer Systems and Communications

Brno, Fall 2025


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:

- ChatGPT to improve my writing style
- ChatGPT and Claude for faster code writing

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Ondřej Marek

Advisor: RNDr. Martin Kuba, Ph.D.

Acknowledgements

I would like to thank my supervisor, RNDr. Martin Kuba, Ph.D., for his guidance and valuable advice. I would also like to thank my consultant, doc. RNDr. Miloslav Zejda, Ph.D., for his expert guidance in astronomy. I also thank my family for their support and patience during the writing of this thesis and throughout my studies.

Abstract

In order to study the variability of stellar objects, it is essential to obtain as many measurements as possible, ideally covering the longest time interval possible. Numerous star surveys produce large amounts of photometric data, which are stored in their respective archives.

However, each project typically uses its own data format. For astrophysicists, searching through these archives and converting the data into a unified format can be time-consuming.

AstroCollector is a web application designed to address this issue. Its main purpose is to search for photometric data across various star survey projects. The collected data can be visualized as a light curve or a phase curve. It is also possible to download the data in both processed and original formats for further use.

Keywords

star surveys, photometry, light curve, Python, FastAPI, TypeScript, React

Contents

List of Figures

Introduction

The aim of this thesis is to develop a web application that searches, collects, and unifies photometric data of variable stars from various astronomical digital archives. The customer is the Department of Theoretical Physics and Astrophysics, Faculty of Science, Masaryk University, which is represented by my thesis consultant, doc. RNDr. Miloslav Zejda, Ph.D.

For the study of variability of stellar objects, it is essential to obtain as many brightness measurements as possible, spanning the longest possible time interval. Numerous photometric surveys produce vast amounts of data that are stored in their respective project archives. Unfortunately, each of these projects typically follows its own conventions, including specific data formats. As a result, it is often time-consuming for astrophysicists to manually browse individual archives, retrieve the relevant data, and convert them into a unified data format.

This application will search, based on a given name or coordinates, for data related to the stellar object in archives of photometric sky surveys. The retrieved data will be downloaded and converted into a standardized format consisting of time in Julian date, brightness in magnitude, and magnitude error. Subsequently, the data will be visualized as a light curve, or in the case of periodically variable objects, as a phase curve.

In the first chapter, we explain core astronomical concepts such as photometric data, time standards, magnitudes, and coordinate systems. It also introduces object-name resolving services and the visualization of brightness measurements as light and phase curves.

The second chapter analyzes available tools. We discuss the customer's functional and non-functional requirements and present the application's use-case diagram.

In the third chapter, we present the overall architecture of the application, divided into client, server, and deployment modules.

The fourth chapter details the server-side part of the application. It describes key implementation aspects such as the plugin system, task management, persistence model, export features, and security.

In the fifth chapter, we discuss the client-side part of the application. We examine the technology stack and implementation of important features, such as data visualization.

The sixth chapter describes the containerized deployment. It also explains the configuration of the services.

In the seventh chapter, we provide a practical guide for creating and integrating a new catalog plugin. The chapter also explains how to upload the plugin and its resources through the administration interface.

The last chapter proposes enhancements of the application. These ideas outline the potential future development paths of the project.

1 Astronomical terminology

In this chapter, I will explain the astronomy-related terms necessary to understand the application.

The key term used in this work is *photometric data*. Photometry is a technique that measures the brightness of a star in an image [1]. The data are measured from both space and ground-based observatories.

We are interested in those parts of the measurements: *time stamp*, *magnitude*, *magnitude error*, and the *photometric filter* used.

Time stamp is usually expressed in *Julian Date*. That is the number of days elapsed since noon on January 1, 4713 BC [2]. However, the time stamp is defined unequivocally by its *time standard* and *reference frame*. The *time standard* refers to the way a particular clock ticks and its arbitrary zero point, as defined by international standards [3]. The *reference frame* is the geometric location from which one could measure time. Different reference frames differ by the light-travel time (LTT) between them [3]. If we want to work with data from multiple sources, we need to unify the time stamps to a common time standard and reference frame.

As suggested in [3], we should aim to produce the time stamps in the $\mathrm{BJD_{TDB}}$ format. BJD (Barycentric Julian Date) is the reference frame, which refers to the barycenter of the Solar System. TDB (Barycentric Dynamical Time) is the time standard: TT (Terrestrial Time) corrected for relativistic effects at the barycenter of the Solar System [4].

Magnitude describes how bright an object appears in the sky from Earth. Astronomers usually measure the flux of an object by collecting light with a telescope and camera. Using *photometric filters*, they select specific regions of wavelengths to determine brightness only in this spectral area. The magnitude of an object is then computed based on the flux and the standard zeroth-magnitude flux for the chosen filter [5].

To collect photometric data of a specific sky region, selected stars, or the entire sky is the aim of many *photometric sky surveys*, however, the provided data are not in a unified format. Usually, the survey archives differ in time stamp formats and also in the access method. Some surveys expose an API, but others must be downloaded and

worked with data locally. For simplicity, I use the term *catalog* instead of photometric sky surveys in this work.

The surveys often have their own way of identifying stellar objects, as they introduce their own identifiers. The reliable cross-catalog way of object identification is to use *celestial coordinates*. The Equatorial Coordinate System uses right ascension and declination. Right ascension (abbreviated RA) is similar to longitude and is measured in hours, minutes, and seconds (or in corresponding degrees) eastward along the celestial equator. Declination is similar to latitude and is measured in degrees, arcminutes, and arcseconds, north or south of the celestial equator [6].

Fortunately, there are services to resolve the commonly used names of stellar objects (for example, Sirius) into their celestial coordinates. The most popular are SIMBAD[7] and VSX[8].

The photometric data can be visualized as a *light curve*, or in the case of periodically variable objects, as a *phase curve*. A *light curve* is a plot showing the brightness variation of a source over time. For periodic sources, it is common to display the brightness variations as a function of phase [9]. Phase is computed as follows:

$$\phi = \mathrm{frac}\left(\frac{\mathrm{JD} - T_0}{P}\right)$$

where

ϕ : phase value,
JD : Julian Date of the observation,
T_0 : reference epoch,
P : period of the object,
$\mathrm{frac}(x)$: fractional part of x ($x - \lfloor x \rfloor$).

2 Current state overview and requirements

For astrophysicists, it is time-consuming to search for relevant photometric data through different star surveys. Currently, there is no such online tool addressing this issue. If someone wants to collect photometric data from multiple sources, they must visit each one individually. Each source typically uses a different data format, and the user must consult the project documentation to understand it. The data then has to be manually converted into a common format by hand.

For these reasons, it has become necessary to create a tool that automates this tedious task for astronomers. The collected data will be available for export, both in converted and original formats.

In this chapter, we examine similar existing applications. Then, we present the customer's requirements and provide a use case diagram based on those requirements.

2.1 Other existing applications

The customer introduced me to an old desktop application called PDR [10], created in 2016. Since then, many star surveys have changed their data formats, making the application less usable. New catalogs have also emerged that are not covered by the application, as it is no longer maintained. Another issue is accessibility, as it is a desktop application that requires installation. Although access to new star surveys can be added through application plugins, this functionality is limited to the local environment.

Based on my market research, several online tools provide access to photometric data:

VizieR

VizieR[11] is a library of published astronomical catalogs. However, the catalogs do not share unified data format, as the aim of the service is to make the catalogs available to the public online. Moreover, it is not user-friendly, as there are many forms and form elements, making the process of gathering photometric data less straightforward.

MAST

The MAST[12] portal is a website that allows users to search through multiple collections of astronomical datasets in one place. The data is not in a unified format and requires further processing.

IRSA

The third example is NASA's IRSA[13]. It allows users to search for data in NASA's infrared and submillimeter missions. Like the two examples above, it also suffers from inconsistent data format.

Evaluation

None of the tools listed above meet the customer's requirements, described in section 2.2.

2.2 Requirements of the customer

Based on discussions with the customer, the following requirements were identified:

2.2.1 Functional requirements

FR1	**Search by object name** The system shall allow users to search for stellar objects by name and resolve the name to sky coordinates using SIMBAD[7] or VSX[8].
FR2	**Search by coordinates** The system shall allow users to search for stellar objects by specifying sky coordinates and a search radius.
FR3	**Display sky map** The system shall display the area surrounding the searched stellar object on a sky map.
FR4	**Fetch photometric data**

	The system shall retrieve photometric data for the selected stellar objects from supported catalogs.
FR5	**Unify data format** The system shall convert the collected data into a unified format: • timestamp in BJD_{TDB} • magnitude • magnitude error • photometric filter
FR6	**Visualize data** The system shall visualize the photometric data as a light curve or a phase curve.
FR7	**Phase curve and light curve data filtering** The system shall allow filtering of data by catalogs or by photometric filters.
FR8	**Export data** The system shall allow exporting either the unified data or the original data.

2.2.2 Non-functional requirements

NFR1	**Web application** The system shall be delivered as a web application accessible through a standard web browser.
NFR2	**Interactive plots** The system shall provide zoomable and pannable plots for data visualization.
NFR3	**High-volume plotting** The plotting functionality shall be able to handle a large number of data points without significant performance degradation.
NFR4	**Containerized deployment**

	The application shall run inside Podman[14] containers.
NFR5	**Automatic nearby-object retrieval** The system shall automatically download photometric data for stellar objects within a radius of 0.05 arcseconds from the searched coordinates. Other nearby stellar objects shall require manual selection.
NFR6	**Trial operation** The system shall undergo one month of trial operation on the server of the Department of Theoretical Physics and Astrophysics, Faculty of Science, Masaryk University.

2.3 Use case diagram

The functionality of the application is shown in the use case diagram 2.1.

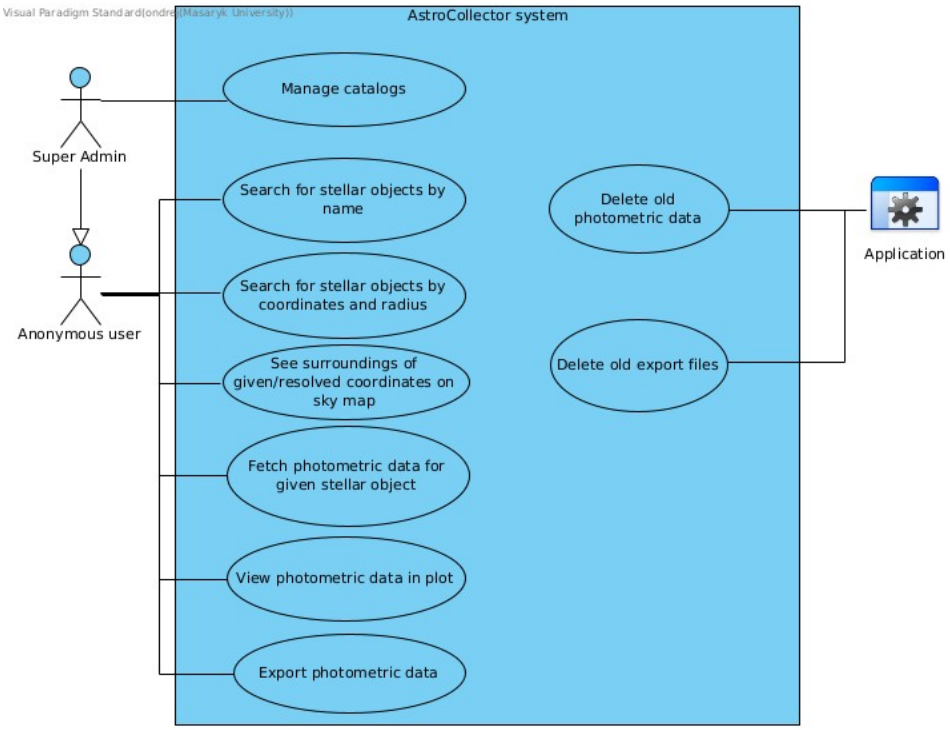

Figure 2.1: Use case diagram of AstroCollector.

There are three actors in the AstroCollector web application: *Anonymous user*, *Super Admin*, and *Application*.

The first actor is the Anonymous user, who represents anyone interacting with the application without authentication. They can use most of the application functionality. Any user can search for stellar objects within the supported catalogs using either a name, or coordinates and radius. When searching by name, it is resolved to coordinates using SIMBAD [7] or VSX [8]. The sky map with close surroundings will be shown for the given coordinates. The Anonymous user can also select various found stellar objects across the catalogs and fetch the corresponding photometric data. The data will be in the unified format — Julian date (in $\mathrm{BJD_{TDB}}$), magnitude, magnitude error, and used photometric filter, if known. It is plotted as a light curve or a phase curve. The plots should be interactive — pannable and zoomable. In addition, they should support filtering by catalogs or photometric filters. The data can be exported on demand, either in the unified format, or in the original format from the source.

The Super Admin has the ability to manage the catalogs. Such user can perform CRUD[1] operations on the catalogs. This enables the application to be extensible, as new data sources can be added, updated or removed during runtime.

The last actor is the *Application* itself. The application schedules periodic jobs to clear the database and export files. Processed data, raw data and export files must be cleaned regularly to free storage space, as over time we will download large amounts of data.

Another necessary requirement was accessibility. The target audience does not want to deal with installing additional software. For that reason, it was decided that AstroCollector will be a web application.

AstroCollector will run on a server at the Department of Theoretical Physics and Astrophysics, Faculty of Science, Masaryk University. The server uses Podman[14] to deploy the application. Therefore, it has to be orchestrated within containerized environment.

1. Create, Read, Update, Delete

3 Application design and architecture

In this chapter, we walk through the application design, which follows the requirements. The project's source code is divided into three parts: the client (frontend) module, the server (backend) module, and the deployment module. The application components are shown in figure 3.1.

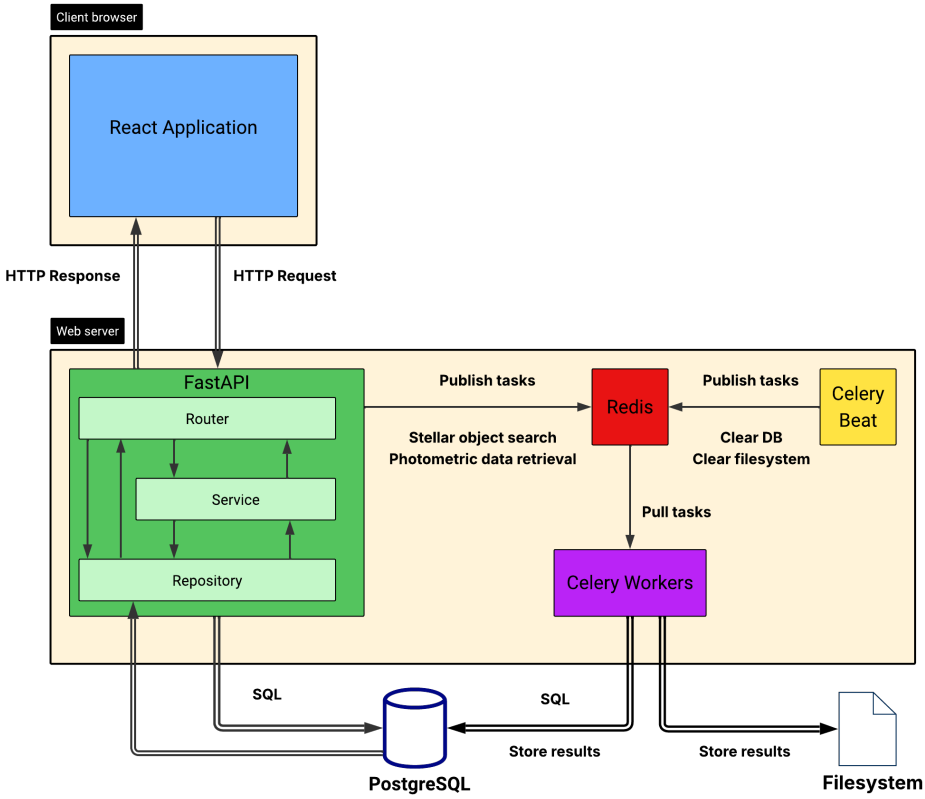

Figure 3.1: AstroCollector components.

AstroCollector follows a client-server architecture. The web server implements a *REST API*[1]. At the heart of the web server is a *FastAPI*[2]

1. *Application Programming Interface* following the *Representational State Transfer* architectural style
2. https://fastapi.tiangolo.com/

application, which handles incoming HTTP[3] requests from the client. The Fast API application delegates the heavy work (searching for stellar objects and fetching photometric data) to *Celery*[4], a distributed task queue, as tasks. The tasks are published to a message broker, a *Redis*[5] instance. *Celery* can then pull those tasks and distribute them among the workers.

Results of the tasks are stored in a *PostgreSQL*[6] database. The client can request the FastAPI application for the results of a completed task to obtain the photometric data in the unified format. Additionally, the photometric data is also stored in a *CSV*[7] file in its original format. PostgreSQL and Fast API or Celery workers communicate via SQL[8].

3.1 Server module

The web server contains all application logic. The FastAPI application is divided into three layers: the presentation layer, the service layer and yhe repository layer. It follows the concept of an *open-layered* architecture, meaning that a layer does not have to communicate only with the layer directly beneath it. Instead, it allows direct access to any layer underneath, which helps to reduce boilerplate. However, it introduces more dependencies.

The FastAPI *routers*[9] form the presentation layer, as they expose the API to the Internet and act as the single entry point to the application. The underlying layer is the service layer, where the business logic is located. The third layer, the repository layer, provides access to the database via SQL for retrieving and storing data.

Another part of the web server is the task queue. Redis acts as the message receiver. The messages are either tasks from the *plugin system* (4.2.2) to search for stellar objects and fetch photometric data, or periodic database and filesystem cleanup scheduled by *Celery Beat*. The API allows the user to enqueue the tasks.

---

3. Hypertext Transfer Protocol
4. https://docs.celeryq.dev/en/stable/
5. https://redis.io/
6. https://www.postgresql.org/
7. Comma-separated values
8. Structured Query Language
9. https://fastapi.tiangolo.com/reference/apirouter/

The core part of the application is the plugin system, as we have to allow modifications (CRUD operations) of the plugins during runtime. Every plugin corresponds to a single catalog, which it integrates into the application and which it can query. Each plugin is a separate module that implements a predefined interface. The interface is simple, as it contains two methods:

- **Search phase** - method for searching identifiers of stellar objects.
- **Data phase** - method to fetch and process photometric data for a given identifier. We also want to store the original data here to avoid having to query the catalog again on export.

The results of the search phase and the data phase are persisted afterwards.

Since the catalogs have different ways of identifying stellar objects, the identifiers have to be flexible as well. For that reason, we have to allow the plugin developer to define their own identifier, which is understood by the corresponding plugin.

During both the *search phase* and the *data phase*, external services (the catalogs) are queried. In some cases, it can take quite long for the catalog to respond. Moreover, the data phase also requires performing timestamp conversions, magnitude and magnitude error computations, or data filtering. To avoid timeouts, I have decided to use a *task queue*.

The task queue is a system for managing background work executed outside the HTTP request-response cycle. This allows us to run the tasks in the *search phase* and the *data phase* asynchronously. Furthermore, we can enqueue tasks periodically to perform storage cleaning.

3.2 Client module

The client side is an SPA[10] implemented in React, running in the user's web browser. The core part of the client application is the page for searching stellar objects, retrieving their corresponding photometric

10. Single-page application

data, and visualizing the results. It also includes a catalog management page for administrators, where they can manage catalog plugins.

3.3 Deployment module

This module contains directories for the application deployment and configuration, both in the production environment and locally. It includes *Compose files*[11] and *.env files*[12] to configure the services. The deployment is explained in chapter 6.

11. https://docs.docker.com/compose/intro/compose-application-model/#the-compose-file
12. https://dotenvx.com/docs/env-file

4 Server side

In this chapter, we discuss the server-side part of the application. Section 4.1 presents the technology stack, and section 4.2 describes the implementation.

4.1 Technologies

4.1.1 Persistence

The persistence needs of the application are fully covered by using a *relational database*[1] in combination with filesystem storage for files. I chose to use PostgreSQL[2], as it is one of the most popular solutions and I had previous experience with it. It is an open-source object-relational database system that offers high performance and robustness.

However, additional resource files and plugin scripts should not be stored in the database. Both plugin scripts and additional resources are stored directly on disk.

4.1.2 API framework

Choosing the right API framework is one of the most important decisions, as it affects how the application is developed. I selected the framework based on the following main criteria:

- developer experience — the maturity of the framework, its ecosystem, documentation, and available resources
- astronomy-related tools ecosystem
- personal familiarity with the technology

Based on my experience, *Spring Boot*[3] or *ASP.NET Core*[4] would be strong candidates. Both are popular, well-established frameworks

1. https://cloud.google.com/learn/what-is-a-relational-database
2. https://www.postgresql.org/
3. https://spring.io/projects/spring-boot
4. https://dotnet.microsoft.com/en-us/apps/aspnet

with rich ecosystems and documentation. They provide built-in infrastructure that simplifies the development process, including database access, security, and *dependency injection*[5]. Spring Boot is written in Java, whereas *ASP.NET Core* is powered by C#. The two frameworks are also comparable in terms of performance, as shown by benchmark results[6].

Since the application deals with the processing of photometric data and other astronomy-related tasks, it is desirable to choose a platform that supports these operations. The most suitable platform is *Python*, as it is widely used in the astronomy community. There are several Python packages dedicated to the astronomy community, such as *Astropy*[*15*], *PyVO*[*16*], or *Astroquery*[*17*].

I decided to use Python as the server-side programming language. Although Java or C# would be a better choice in terms of robustness and type safety, the mentioned Python packages are essential and the most important criterion for this application. There are multiple Python web frameworks, such as *Flask*[7], *FastAPI*, or *Django*[8]. I chose FastAPI, as it provides a good balance between flexibility and structure. Flask is too lightweight and lacks necessary infrastructure, whereas Django is more suitable for full-stack applications.

FastAPI is a modern web framework for building APIs. It supports asynchronous programming by default, provides automatic OpenAPI[9] documentation for the endpoints, and offers solid performance[10]. Its performance is slightly lower than that of Spring Boot or ASP.NET Core; however, raw performance is not a high priority compared to the other criteria. While its ecosystem is not as extensive as those of Spring Boot or ASP.NET Core, it is fully sufficient for the requirements of our application.

5. https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection
6. https://www.techempower.com/benchmarks/
7. https://flask.palletsprojects.com/en/stable/
8. https://www.djangoproject.com/
9. https://www.openapis.org/
10. https://www.techempower.com/benchmarks/

4.1.3 Task queue

As explained earlier, we want to run code asynchronously outside of the HTTP request-response cycle. Therefore, a task queue was required. There are plenty of suitable candidates, such as *Celery*[11], *RQ*[12], or *Dramatiq*[13]. I decided to use Celery, as it is a well-established and reliable solution. One of its main advantages is that it comes with a scheduler process for periodic tasks, *Celery beat*[14].

Celery Beat schedules cleanup tasks to free up space in the database occupied by old photometric data. This refers to data fetched from the catalogs that has exceeded the expiration time 4.2.7. The same task also clears expired export archive files and CSV files containing the original photometric data.

4.1.4 Validation

Validation of request input data and *DTOs*[15] is handled by the *Pydantic*[16] library. Pydantic allows defining *models*. Models are classes that inherit from `BaseModel` and define fields as annotated attributes [18].

4.1.5 Database access and ORM

The project uses *SQLAlchemy*[17] to access the database. It provides an SQL toolkit and Object Relational Mapper (ORM) in Python.

4.2 Implementation

In this chapter, we discuss the implementation details of the server side.

11. https://docs.celeryq.dev/en/stable/
12. https://python-rq.org/
13. https://dramatiq.io/
14. https://docs.celeryq.dev/en/latest/userguide/periodic-tasks.html
15. Data transfer object - an object carrying data between layers or processes
16. https://docs.pydantic.dev/latest/
17. https://www.sqlalchemy.org/

4.2.1 Directory structure

Figure 4.1: Important directories of the backend part

Figure 4.1 shows the directory structure of the backend part of the application. FastAPI is flexible and does not require developers to follow a predefined directory structure.

The *alembic* directory contains database migrations created by the *Alembic*[18] tool. The migrations are described later in subsection 4.2.4.

Application logs are stored in the *logs* directory. It includes logs from both the FastAPI application and Celery.

The *plugins* directory contains the catalog plugins currently in use by the application. This directory is a *Python package*[19]. During runtime, files in this directory may be added or removed. Plugins are discussed in greater detail in subsection 4.2.2.

Additional plugin resources are stored in the *resources* directory, as some catalogs need to be downloaded and searched locally.

The *temp* directory stores temporary files, which are periodically deleted. It stores CSV files containing original data fetched from catalogs, as well as exported archive files. It can also be used by any plugin or other part of the application that requires temporary storage.

Lastly, the *src* directory is a Python package that contains the source code of the application. It follows the *vertical slice architecture* approach, which groups source code files by features[20].

The *core* package contains shared infrastructure used across features:

- The *celery* package provides the Celery application setup and configuration.
- The *config* package contains the application configuration, which is further described in subsection 4.2.7.
- The *database* package contains the database connection setup.
- The *exception* package defines custom application exceptions.
- The *repository* package contains a generic repository implementation, which is described in detail in subsection 4.2.4.
- The *service* package provides DTOs used by the service layer.

18. https://alembic.sqlalchemy.org/en/latest/
19. https://packaging.python.org/en/latest/tutorials/packaging-projects/
20. https://www.jimmybogard.com/vertical-slice-architecture/

- Authentication and authorization-related functionality is implemented in the *security* package, which is described in subsection 4.2.6.

The *data_retrieval* package implements the logic for retrieving task results, such as found stellar objects or processed photometric data.

The *export* package provides data export functionality. This is described in detail in the subsection 4.2.5.

The *phase_curve* package contains functionality to fetch the phase curve parameters (epoch and period) from the VSX catalog.

The *plugin* package provides the plugin management functionality for CRUD operations, resource uploading, and plugin scripts uploading and downloading. It contains the following packages.

- The *interface* package defines the catalog plugin interface and associated DTOs.
- The application comes with pre-implemented catalogs stored in the *default_plugins* package. Each subpackage corresponds to one catalog plugin. Details of the plugin system are described in the subsection 4.2.2.

The *so_name_resolving* package contains the functionality for resolving the names of stellar objects to coordinates.

The *tasks* package is responsible for enqueuing tasks and querying their status. This is described in detail in the subsection 4.2.3.

The *main.py* file serves as the entry point of the application. This *Python module*[21] defines the `FastAPI` instance and handles startup initialization, such as database seeding[22], logging setup, exception handler registration, and router registration.

4.2.2 Plugin system

This subsection describes the details of the core part of the system — the *catalog plugins*.

Every plugin consists of three parts: a plugin script, a correspoding plugin database entity to store metadata, and a resources directory. The database keeps track of the registered plugins.

21. https://docs.python.org/3/tutorial/modules.html
22. Populating a database with an initial set of data

Every plugin must implement a class that extends the `CatalogPlugin` abstract class defined in the `src.plugin.interface` package. The plugin contract is shown in the code snippet 4.1.

Listing 4.1: Plugin contract

```
T = TypeVar("T", bound=StellarObjectIdentificatorDto
    )

class CatalogPlugin(Generic[T], ABC):

    @abstractmethod
    def list_objects(
            self,
            coords: SkyCoord,
            radius_arcsec: float,
            plugin_id: UUID,
            resources_dir: Path,
    ) -> Iterator[List[T]]:
        pass

    @abstractmethod
    def get_photometric_data(
            self, identificator: T, csv_path: Path,
               resources_dir: Path
    ) -> Iterator[list[PhotometricDataDto]]:
        pass
```

The first method is the `list_objects`. It must return stellar objects that were found in the radius around the given coordinates. The stellar objects are identified by the `StellarObjectIdentificatorDto` subclass.

The second method `get_photometric_data` is responsible for fetching photometric data for a given stellar object. The fetched data must be converted into the unified format as `PhotometricDataDto` objects before being returned. The original data fetched from the catalog must be written to the CSV file specified by the `csv_path` parameter.

Both methods include the `resources_dir` parameter, which provides the access to the plugin's corresponding resources directory. The

methods are also *generator functions*[23]. In the case of large datasets, we want to be able to store them in the database in chunks rather than all at once. For this reason, we use generators, as they allow us to *yield* the data in chunks.

Each catalog has its own way of identifying stellar objects. To make the system flexible and allow every catalog to manage the identification independently, the plugin class is parametrized by a subclass of `StellarObjectIdentificatorDto`. This enables the developer to add an arbitrary number of attributes that is needed to identify a stellar object within the catalog.

The process of adding a new catalog is described in chapter 7. The application comes with pre-implemented plugins in the `src.plugin.default_plugins` package. The plugins extend the `DefaultCatalogPlugin` class, which is a subclass of `CatalogPlugin`. The difference between them is that the catalog metadata is included directly in the code, so that the plugins can be automatically added on application startup.

Since plugins can be dynamically added, they must also be imported dynamically at runtime. This is achieved using the `importlib`[24] package, which is able to import the `CatalogPlugin` subclass from the Python module.

4.2.3 Task management

Searching for stellar objects and data fetching are performed asynchronously by Celery workers. Celery is composed of three building blocks [19]:

- *Producer*: The part of the application that requests a task for asynchronous execution. For example, `task1.delay(a=1, b=2)` in an endpoint.
- *Message Broker*: The storage system to which the producer writes the serialized task. In our case, this is Redis.
- *Worker*: A unit of work that subscribes to new messages on the message broker and executes them.

23. https://realpython.com/introduction-to-python-generators/
24. https://docs.python.org/3/library/importlib.html

We can choose which underlying concurrency model the worker uses to execute tasks. The options are:

- processes,
- coroutines, and
- threads.

For this use case, I chose to run the tasks in separate processes. This is because the tasks typically involve computations, as the plugin needs to convert the timestamps into the BJD_{TDB} format. Initially, I tried to implement this using FastAPI's Background tasks[25]. However, after testing, I found that this approach was not sufficient for the use case. It caused the application to become unresponsive, so I introduced a task queue instead.

The tasks are defined in the `src.tasks.tasks` module and can be enqueued using endpoints in the `src.tasks.router` module. There is also an endpoint that returns the current status of a task. When a task is finished, we can use its ID to retrieve its result data.

This approach requires the client to use *polling* to retrieve data. Polling is a simple technique in which the server is queried at regular intervals about the status of the task.

25. https://fastapi.tiangolo.com/tutorial/background-tasks/

4.2.4 Persistence and data access

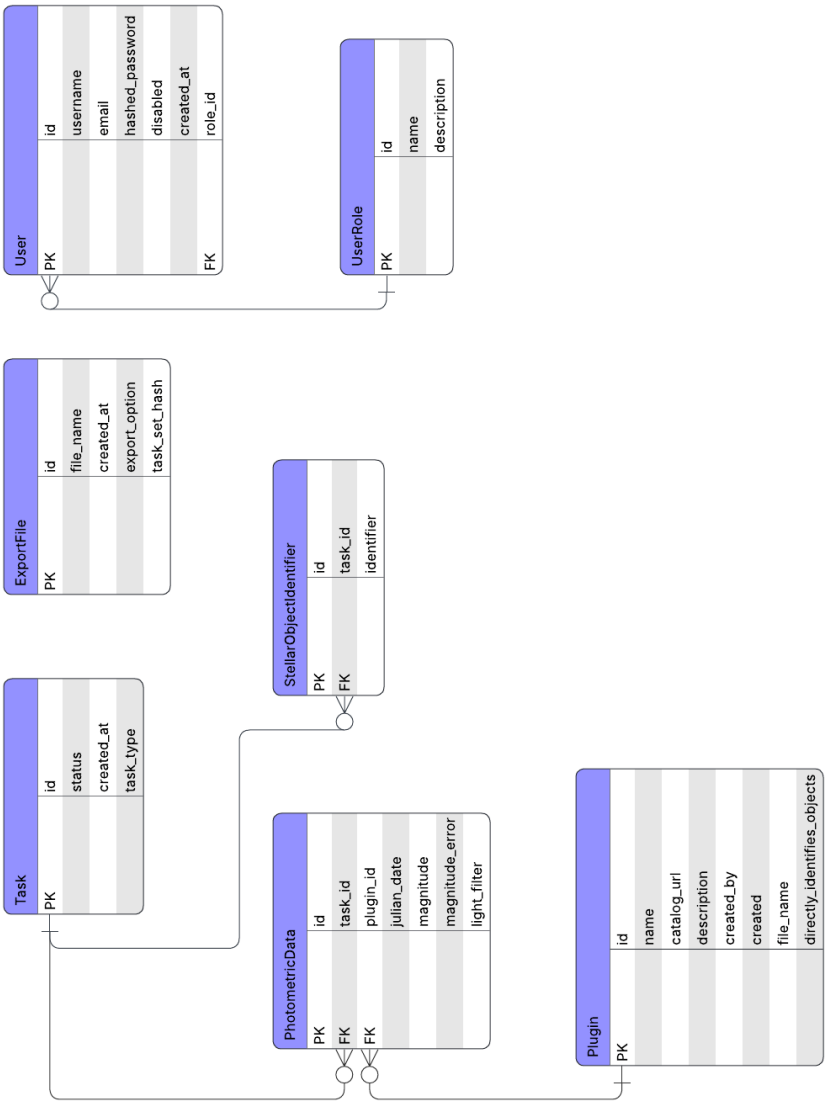

Figure 4.2: Entity relationship diagram

The entity-relationship diagram of AstroCollector is shown in figure 4.2.

The cornerstone of the application is the catalog plugin system. The *Plugin* entity represents a catalog plugin in the application. The `directly_identifies_objects` flag indicates whether the catalog groups photometric data by stellar object identifiers. The `file_name` field stores the filename of the associated script located in the *plugins* directory.

The *Task* entity tracks search and photometric data fetching tasks. It is linked to two entities, *StellarObjectIdentifier* and *PhotometricData*, which represent the task results. The StellarObjectIdentifier entity represents an identifier obtained using the `CatalogPlugin.list_objects()` method. The identifier itself is stored in the `identifier` field as *JSON*[26]. The PhotometricData entity represents processed photometric data in the unified format.

The *ExportFile* entity represents an export archive file that consists of data from multiple tasks. The field `task_set_hash` identifies the ExportFile entity based on the associated Tasks. The `export_option` field specifies the export option used to generate the archive. Export is described in detail in subsection 4.2.5.

An important field in both the *ExportFile* and *Task* entities is `created_at`. It is used by the cleanup task to determine whether the record has expired and should be deleted.

The *User* entity stores information about the system users. It is linked to the *UserRole* entity, as every user is assigned a role.

To manage database schema changes, I use Alembic. It allows us to track the history of changes and roll back to previous versions of the schema.

The repository layer in the application is responsible for accessing data. The package `src.core.repository` contains an implementation of a generic repository using SQLAlchemy. It provides support for CRUD and simple filtering.

4.2.5 Export

Photometric data can be exported as a *ZIP file*[27] containing CSV files. We can export:

---

26. JavaScript Object Notation
27. https://www.iana.org/assignments/media-types/application/zip

- raw data
- processed data (in the unified format). These data can either be grouped by their sources (catalogs) and split into multiple files, or included together in a single file.

When requesting an export, we must include the task IDs from which the data originate. Based on these task IDs, a `task_set_hash` value is computed, which is used to identify the corresponding *ExportFile* entity. If the same export file is requested later (using the same task IDs), we can compute the hash and skip creating a new file if one already exists.

To avoid running out of disk space, created archives are available only for a limited time. When this time expires, the archive is deleted from the *temp* directory along with its corresponding *ExportFile* record.

4.2.6 Security

The API uses session-based authentication. Upon successful login, the server establishes a user *session*.

A random session ID is generated by the server, stored in a Redis in-memory database, and associated with the authenticated user. This session ID is then sent to the client as a cookie, protected using the `HttpOnly` and `Secure` attributes. Additional security measures include specifying the cookie's `Domain` and `SameSite` attributes [20].

For subsequent requests, the client includes the session cookie, and the request is processed only if the corresponding session exists. Sessions have expiration times, which is currently set to one day.

Using cookies for authentication introduces a vulnerability known as *CSRF* (Cross-Site Request Forgery). A description of how CSRF attacks work is available in [21]. To mitigate this risk, the application generates a *CSRF token* (a random string) after successful authentication. This token is stored in the session data on the server and returned to the client. On the client side, the token is stored in the *local storage*[28] and attached to every request using the `X-CSRF-Token` header. For each such request, the server verifies that the received CSRF token matches the value stored in the session.

28. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

Upon logout or session expiration, the session data is deleted from Redis. This action revokes the session ID, which can no longer be used to authenticate subsequent requests.

The application uses role-based access control, implemented by the `required_roles` function. After validating the session and the CSRF token, the application loads the user's role from the database and checks it against the allowed roles. Only users with the appropriate role may access protected endpoints. Currently, only plugin-related endpoints are protected, as the remaining endpoints are intended for public use. Listing 4.2 shows an example of protecting the delete plugin endpoint.

Listing 4.2: Endpoint protection

```
@router.delete("/{plugin_id}")
async def delete_plugin(
    _: Annotated[User, Depends(required_roles(
       UserRoleEnum.super_admin))],
    plugin_id: UUID,
    service: PluginServiceDep,
):
    # ... implementation
    pass
```

Only the super admin role is allowed to access this endpoint.

In the *main.py* file, the CORS[29] policy is defined, enabling proper communication between the frontend development server and the backend server.

4.2.7 Configuration

To define the configuration file, we use the *pydantic-settings*[30] package. The `src.core.config.config` module contains the `settings` object, which holds the application's configuration. It defines database connection properties, such as the user and password. It includes paths to important directories, such as the *plugin* and *temp* directories. It defines the logging configuration and Celery configuration. Moreover,

29. Cross-origin resource sharing
30. https://docs.pydantic.dev/latest/concepts/pydantic_settings/

it contains the security configuration, including session-expiration times and cookie attributes for the session cookie.

Configuration values can be provided via the `.env` file in the project's root directory.

5 Client side

In this chapter, we discuss the client-side part of the application. First, we examine the technology stack in section 5.1. Then, we describe the implementation in section 5.2.

5.1 Technologies

5.1.1 JavaScript framework

JavaScript has become the leading technology for developing modern user interfaces, with the advent of single-page applications. JavaScript is used to dynamically update the page in response to user input.

This led to the emergence of multiple JavaScript frontend frameworks, such as *React*[1], *Vue.js*[2] or *Svelte*[3]. These frameworks provide developers with pre-built components and tools that accelerate the development.

A key requirement was to choose a well-established and actively maintained JavaScript framework with a large ecosystem of libraries, community resources, and third-party tooling. This narrowed my choice to React and Vue.js. When choosing the framework, I wanted one with support for plotting libraries, such as *Plotly.js*[4] or *Recharts*[5].

Based on my previous experience, I chose React. React also offers seamless integration with both mentioned libraries. I decided to use *TypeScript*[6] as the programming language. TypeScript is a superset of JavaScript that offers strong typing.

5.1.2 Routing

Routing is the backbone of any single-page application, as it provides the illusion of a "regular" web page by using routes in the URL. I chose

1. https://react.dev/
2. https://vuejs.org/
3. https://svelte.dev/
4. https://plotly.com/javascript/
5. https://recharts.org/
6. https://www.typescriptlang.org/

to use *TanStack Router*[7], which offers type-safe routing for React applications.

5.1.3 Data fetching and management

Almost every application needs to fetch remote data, and ours is no exception. However, because the data change over time, the application data must be managed and updated accordingly — a task that is often error-prone and cumbersome. Fortunately, there is a library designed specifically for this purpose: *TanStack Query*[8]. It provides features such as caching, retries, and polling. The motivation for using this library is described in [22].

The application uses *Axios*[9] in combination with TanStack Query to perform HTTP requests. Axios is an HTTP client capable of sending *XHR*[10] from the browser. It simplifies authentication by the allowing automatic inclusion of the session cookie in every request. Moreover, it supports the use of *interceptors*. An interceptor is a callback function, which enables developers to customize the request just before it is sent. This functionality is used to include the CSRF token in each request.

5.1.4 Components

React relies on the concept of components — reusable parts of the user interface. To simplify the development process, I use the *ShadCn*[11] library. It provides a set of components that can be easily customized. For styling the user interface, the application uses *TailwindCSS*[12]. It is a *CSS*[13] framework that provides utility classes that can be composed to create custom designs.

7. https://tanstack.com/router/latest
8. https://tanstack.com/query/latest
9. https://axios-http.com/docs/intro
10. XMLHttpRequest
11. https://ui.shadcn.com/examples/authentication
12. https://tailwindcss.com/
13. Cascading Style Sheets

5.1.5 Forms

Forms are necessary part of our application. To simplify form state management and validation, we use *React Hook Form*[14]. React Hook Form supports *Zod*[15], a TypeScript schema validation library.

5.1.6 Data visualization

The application must be able to visualize the data in a plot. Plotly.js initially seemed to be the most suitable library, as it provides interactive plots. However, when working with large datasets, the application became unresponsive. For that reason, I decided to implement my own solution based on *deck.gl*[16]. The implementation is described in detail in 5.2.3.

5.2 Implementation

In this chapter, we discuss implementation details of the client side. The application is built using *Vite*[17]. Vite is a build tool with a development server that bundles the application using *Rollup*[18].

5.2.1 Directory structure

Figure 5.1 shows the directory structure of the frontend part of the application.

14. https://react-hook-form.com/
15. https://zod.dev/
16. https://deck.gl/
17. https://vite.dev/
18. https://rollupjs.org/

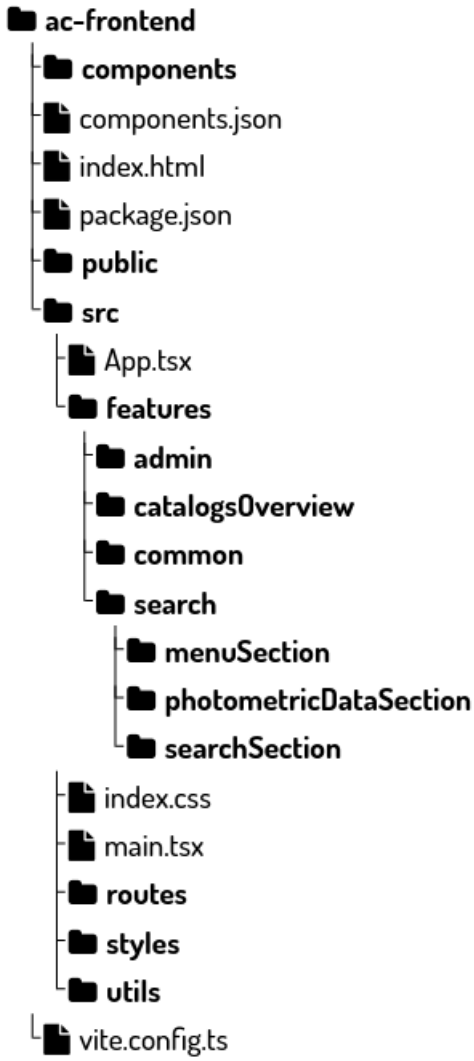

Figure 5.1: Important directories of the frontend part

The core files consist of `index.html`, `package.json`, `main.tsx` and `vite.config.ts`.

The application is contained in the `index.html` file, which references the JavaScript entry point `main.tsx` via the `<script type="module" src="/src/main.tsx"></script>` tag.

The `package.json` file contains all dependencies and scripts, for building the application and running the development server. It is used by *npm*[19], a JavaScript package manager and package registry.

The `vite.config.ts` file contains Vite configuration, such as registering the React plugin or the Tailwind plugin for Vite.

The `components` directory contains ShadCn components used in the project. The `components.json` contains the component configuration.

The `public` directory contains static assets, such as `favicon.ico`.

In the `src` directory, the main component can be found in the `App.tsx` file. The `index.css` file contains the Tailwind configuration. The `routes` directory defines the application routes, which is required by TanStack Router.

The `features` directory is organized using the vertical slice structure. Each subdirectory corresponds to a page, except for the `common` directory, which contains shared functionality, such as error alert components or the HTTP client. The `search` directory corresponds to the main page.

5.2.2 Main page

The main page consists of three parts: the *search section*, *search results section*, and *photometric data section*.

19. https://www.npmjs.com/

Search section

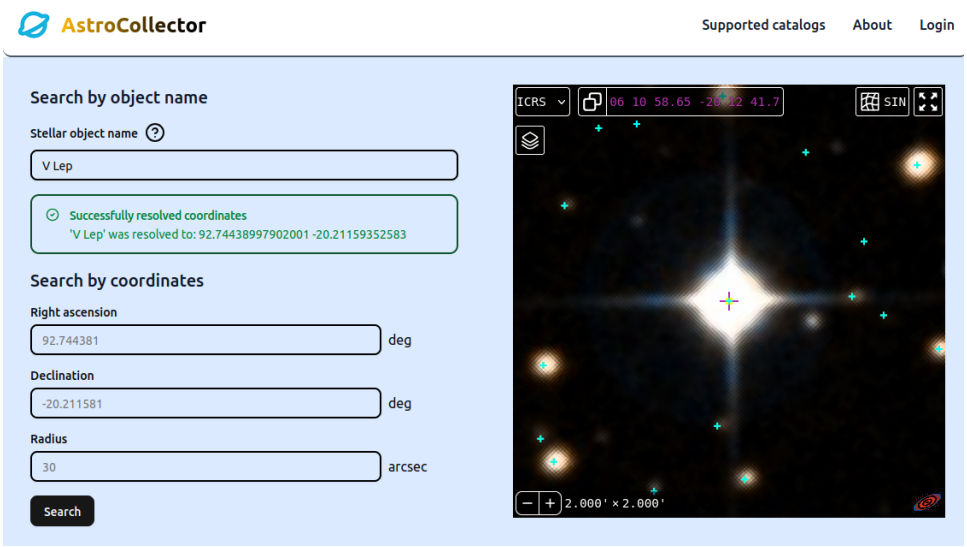

Figure 5.2: Search section

The search section contains a simple form where the user can enter either a stellar object name or coordinates and a radius. A sky map of the surrounding area is displayed using the *Aladin Sky Atlas*[20]. Figure 5.2 shows the search section.

Search results section

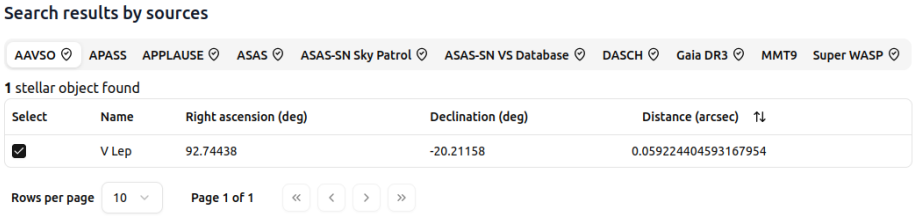

Figure 5.3: Search results section

20. https://aladin.cds.unistra.fr/AladinLite/

This section displays the search results for each catalog. The stellar objects found in each catalog are listed in a table, where they can be selected by the user. The table displays basic information for each record, such as right ascension, declination, and catalog name. All stellar objects within 0.1 arcseconds from the searched coordinates are automatically selected. Figure 5.3 shows the search results section.

Photometric data section

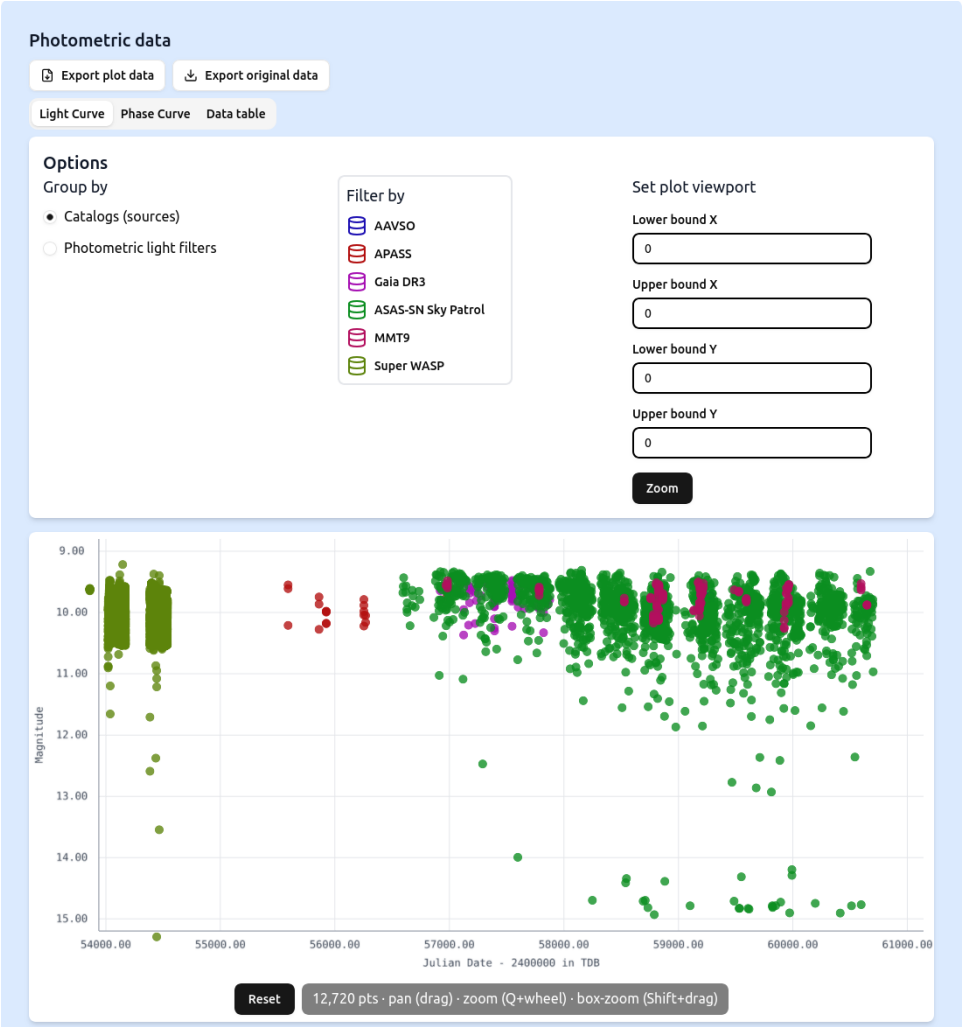

Figure 5.4: Photometric data section

This section is responsible for data fetching and visualization. After the user selects a stellar object in the search section, the corresponding data are downloaded. The gathered photometric data can be displayed as a light curve or a phase curve. Moreover, it contains the control panel, where the user can group the data by the source or photometric filters, toggle data from each group on or off, and set the exact viewport of the plot. The section also contains export buttons for both processed and original data. Figure 5.4 shows the search results section.

5.2.3 Data Plots

Since the application gathers data from multiple sources, it must be able to handle large datasets. When testing the Plotly.js solution with 100 000 points, the application became unusable, as it became unresponsive. I tried testing other plotting libraries, such as:

- *Echarts*[21]
- *regl-scatterplot*[22]
- *ChartJS*[23]

However, none of the solutions met my requirements. In most cases, this was due to performance issues, except for *regl-scatterplot*. The issue with that library was the absence of x- and y-axes and limited customizability.

This led me to discover *deck.gl*, a *WebGL2*[24]-powered visualization framework. The framework achieves high performance by taking advantage of the hardware graphics acceleration provided by the user's device [23]. Deck.gl is based on the concept of visual layers, which can be composed. The layer model is described in detail in [24].

The plot component in the application consists of two layers: `ScatterplotLayer` and `AxesGridLayer`. The `ScatterplotLayer` is responsible for displaying the data points as circles. It also handles data filtering based on the selected display option and selected groups. The

21. https://www.npmjs.com/package/echarts-for-react
22. https://github.com/flekschas/regl-scatterplot
23. https://react-chartjs-2.js.org/
24. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API#webgl_2

`AxesGridLayer` is responsible for displaying the x- and y-axes, as well as the grid lines running across the plot.

5.2.4 Application state management

Although TanStack Query manages the state of remote data, we still need to keep track of other parts of the application state. One such example is user data, which may be needed in different parts of the application. A straightforward solution is to "lift the state up" to the closest common parent component in the hierarchy; however, this pollutes components lower in the tree with unnecessary props—they only pass them further down but do not use them. This situation is known as *prop drilling*.

To solve this issue, we use *Context*. It provides a way to pass data through the component tree without manually passing props at every level[25]. Some of the important application contexts are:

- `AuthContext` — stores the current user's data
- `IdentifiersContext` — stores selected stellar object identifiers from the search section
- `SearchFormContext` — stores the search query from the search form (stellar object name or coordinates)

An alternative to the Context is using a state-management library, such as *Zustand*[26]. However, I decided that Context is sufficient for our use cases.

5.2.5 Catalog management page

The application can be extended with new catalogs. This page, shown in figure 5.5, is accessible only to users with the Super Admin role, who can edit the plugins. It contains all the functionality needed to perform CRUD operations on the plugins. Adding a new catalog is discussed in chapter 7.

25. https://legacy.reactjs.org/docs/context.html
26. https://zustand.docs.pmnd.rs/getting-started/introduction

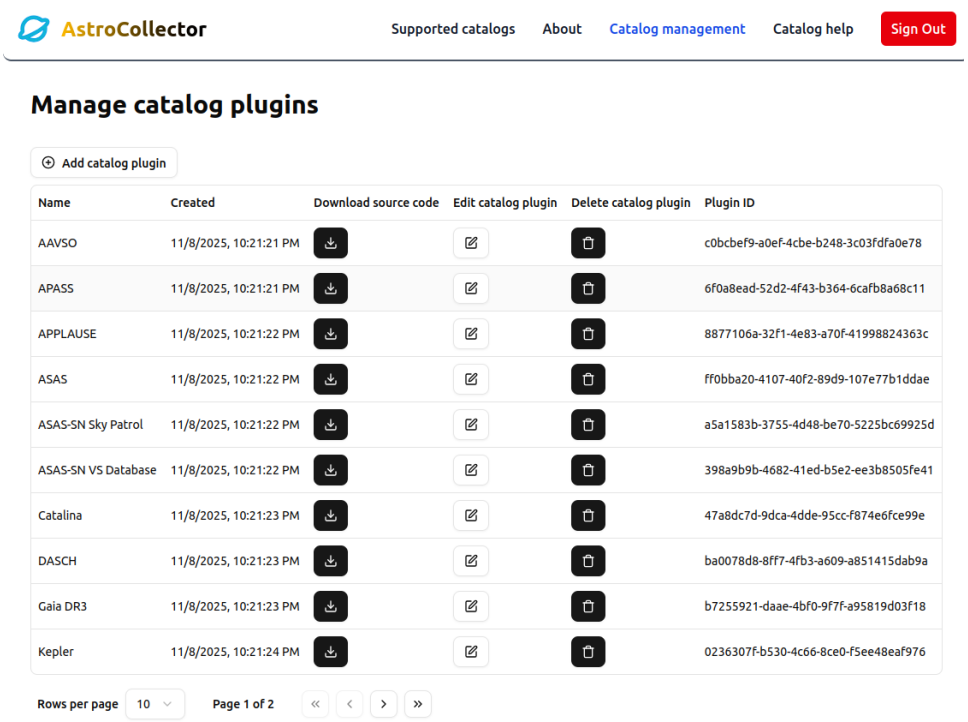

Figure 5.5: Catalog management

6 Deployment

In this chapter, we discuss the deployment of the application. First, we examine the deployed services. Then, we discuss the services configuration.

The services are deployed as *Podman containers*, which are defined in a `compose.yml` configuration file. The `deployment` directory in the project root contains a `compose.yml` file with the following services:

- PostgreSQL instance
- API instance, based off a Python image
- Celery worker instance, based off a Python image
- Celery beat instance, based off a Python image
- Redis instance used as a message broker
- Redis instance used as in-memory database
- *nginx*[1] instance, which serves the single-page application

The backend services operate on the same network, allowing them to communicate with each other. In addition, shared volumes are defined so that the API and Celery services can access the directory with the plugin scripts.

To build the API and both Celery images, a *Dockerfile* is required. A Dockerfile is essentially a text file containing build instructions. As the base, we use a *Python 3.13* image and install the dependencies using the *uv*[2] package manager.

The frontend image is also built using a custom Dockerfile, as the dependencies must be installed using the npm package manager, and nginx configuration is provided to properly serve the application.

At the time of writing, the application is available at `https://staging02.physics.muni.cz/`.

1. https://nginx.org/
2. https://docs.astral.sh/uv/

6.1 Service configuration

The services can be configured using the `.env` file. The developer can configure important variables here, such as the database username and password. A production mode can be enabled here to hide detailed error message responses or the API port can be changed. It is also important to specify the paths to the directories on the server where the plugin resources and logs will be stored.

7 Adding new catalog plugin

To add a new catalog plugin, one must create a Python module that contains two classes:

- a subclass of `StellarObjectIdentificatorDto`
- a subclass of `CatalogPlugin`

The catalog plugin subclass must implement the methods described in subsection 4.2.2. The `StellarObjectIdentificatorDto` subclass can be extended with any additional attributes needed for object identification.

The developer can use popular astronomy-related packages such as `astropy`, `astroquery`, or `pyvo`. The `CatalogPlugin` class contains the utility method `_to_bjd_tdb()`, which can be used by the plugin subclass to convert the timestamps into the BJD_{TDB} format.

The process of adding a new catalog is quite simple. The user must access the Catalog management page and click on the "Add new catalog" button, which can be seen in figure 5.5. This opens the dialog shown in figure 7.1.

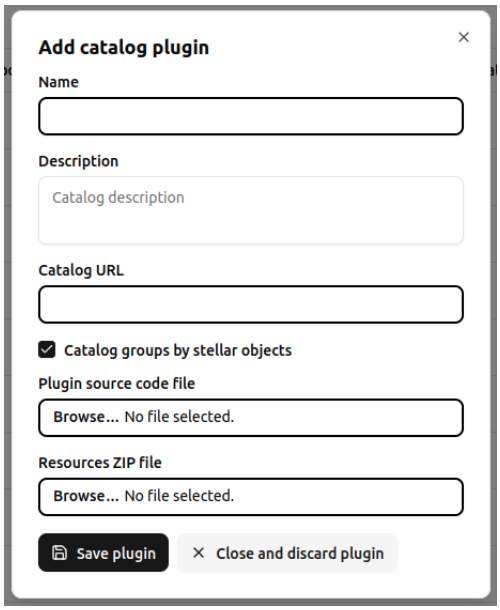

Figure 7.1: Add new catalog plugin

Then, the user has to fill in the catalog metadata, such as the name or the description. The newly created Python module must be uploaded, along with a single ZIP file containing the resources required for the plugin to function properly. The user must keep in mind that the contents of the ZIP file are extracted into the plugin's corresponding resource directory.

After clicking the "Save plugin" button, the files are uploaded to the server and the plugin is ready to be used.

8 Future improvements

In this chapter, I discuss potential enhancements to the application.

I plan to maintain the application in the future. Looking back at the current version, there are several features that could be added. Some of the examples are:

- adding support for additional catalogs
- computing the period from the gathered photometric data
- importing user-provided photometric data
- adding more plot filters, such as filtering by magnitude or timestamp

By adding the new features, the usability and user experience of the application could be further improved.

Conclusion

The aim of this thesis was to develop a web application for searching, collecting, and unifying photometric data of stellar objects from multiple astronomical archives. The resulting tool, AstroCollector, addresses the current situation in which astronomers must manually retrieve data from separate surveys, interpret differing formats, and unify them by hand. AstroCollector automates this workflow by searching for data in supported sky survey archives, converting the data into the homogenized format, and visualizing the results as light or phase curves. It provides an extensible plugin system that allows new data sources to be added as needed.

The application has been running in trial operation, during which user feedback was collected and incorporated into the current version.

AstroCollector was presented in the student session at the 57th Conference on Variable Star Research (Astronomical Institute CAS, Ondřejov, Czechia, November 7–9, 2025)[1].

The complete source code is publicly available in the GitHub repository at `https://github.com/0-mar/AstroCollector`.

1. https://sites.google.com/astronomie.cz/57-konference/o-konferenci

Bibliography

1. OBSERVATORY, Las Cumbres. *Photometry and CCDs* [online]. [visited on 2025-11-22]. Available from: https://lco.global/spacebook/telescopes/what-is-photometry/.
2. ZEJDA, Miloslav. *Fotometrické přehlídky, zdroje dat, čas* [online]. 2019. [visited on 2025-11-22]. Available from: https://www.physics.muni.cz/~zejda/PHV_surveys_casy2019.pdf.
3. EASTMAN, Jason; SIVERD, Robert; GAUDI, B. Scott. Achieving Better Than 1 Minute Accuracy in the Heliocentric and Barycentric Julian Dates. 2010, vol. 122, no. 894, p. 935. Available from DOI: 10.1086/655938.
4. JURYŠEK, Jakub. *Data z robotických přehlídek* [online]. 2018. [visited on 2025-11-22]. Available from: https://www.fzu.cz/~jurysek/lectures/sphe/roboticke_prehlidky_jurysek_2018.pdf.
5. WRIGHT, Edward L. *Magnitudes and Colors* [online]. 2004. [visited on 2025-11-22]. Available from: https://www.astro.ucla.edu/~wright/magcolor.htm.
6. OBSERVATORY, Las Cumbres. *Cosmic Coordinates* [online]. [visited on 2025-11-22]. Available from: https://lco.global/spacebook/sky/equatorial-coordinate-system/.
7. WENGER, M.; OCHSENBEIN, F.; EGRET, D.; DUBOIS, P.; BONNAREL, F.; BORDE, S.; GENOVA, F.; JASNIEWICZ, G.; LALOË, S.; LESTEVEN, S.; MONIER, R. The SIMBAD astronomical database. The CDS reference database for astronomical objects. 2000, vol. 143, pp. 9–22. Available from DOI: 10.1051/aas:2000332.
8. WATSON, C. L.; HENDEN, A. A.; PRICE, A. The International Variable Star Index (VSX). *Society for Astronomical Sciences Annual Symposium*. 2006, vol. 25, p. 47.
9. DUTKEVITCH, Diane. *Light Curves and Phase* [online]. 1998. [visited on 2025-11-22]. Available from: https://astro.wku.edu/labs/m100/phase.html.

10. KRAJČOVIČ, Michal. *Nástroj pro vyhledání fotometrických dat stelárních objektů* [*online*]. 2016 [cit. 2025-11-03]. Available also from: `https://is.muni.cz/th/z0wh7/`. SUPERVISOR : Martin Kuba.

11. AL, Ochsenbein F. et. *The VizieR database of astronomical catalogues*. [N.d.]. Available from DOI: `10.26093/cds/vizier`.

12. *Mikulski Archive for Space Telescopes* (*MAST*) *Portal* [`https://mast.stsci.edu/portal/Mashup/Clients/Mast/Portal.html`]. [N.d.]. [visited on 2025-11-03]. Space Telescope Science Institute (STScI). Accessed: 2025-11-03.

13. *NASA/IPAC Infrared Science Archive* (*IRSA*) [`https://irsa.ipac.caltech.edu/`]. [N.d.]. [visited on 2025-11-03]. California Institute of Technology; NASA archive. Accessed: 2025-11-03.

14. RED HAT, INC. *Podman: A tool for managing OCI containers and pods*. 2025. Available also from: `https://podman.io/`. Accessed: 2025-11-11.

15. THE ASTROPY COLLABORATION; ROBITAILLE, THOMAS P.; TOLLERUD, ERIK J.; GREENFIELD, PERRY; DROETTBOOM, MICHAEL; BRAY, ERIK; ALDCROFT, TOM; DAVIS, MATT; GINSBURG, ADAM; PRICE-WHELAN, ADRIAN M.; KERZENDORF, WOLFGANG E.; CONLEY, ALEXANDER; CRIGHTON, NEIL; BARBARY, KYLE; MUNA, DEMITRI; FERGUSON, HENRY; GROLLIER, FRÉDÉRIC; PARIKH, MADHURA M.; NAIR, PRASANTH H.; GÜNTHER, HANS M.; DEIL, CHRISTOPH; WOILLEZ, JULIEN; CONSEIL, SIMON; KRAMER, ROBAN; TURNER, JAMES E. H.; SINGER, LEO; FOX, RYAN; WEAVER, BENJAMIN A.; ZABALZA, VICTOR; EDWARDS, ZACHARY I.; AZALEE BOSTROEM, K.; BURKE, D. J.; CASEY, ANDREW R.; CRAWFORD, STEVEN M.; DENCHEVA, NADIA; ELY, JUSTIN; JENNESS, TIM; LABRIE, KATHLEEN; LIM, PEY LIAN; PIERFEDERICI, FRANCESCO; PONTZEN, ANDREW; PTAK, ANDY; REFSDAL, BRIAN; SERVILLAT, MATHIEU; STREICHER, OLE. Astropy: A community Python package for astronomy. *AA*. 2013, vol. 558, A33. Available from DOI: `10.1051/0004-6361/201322068`.

16. GRAHAM, Matthew; PLANTE, Ray; TODY, Doug; FITZPATRICK, Mike. *PyVO: Python access to the Virtual Observatory* [Astrophysics

Source Code Library, record ascl:1402.004]. 2014. Available from ascl: 1402.004.

17. GINSBURG, Adam; SIPŐCZ, Brigitta M.; BRASSEUR, C. E.; COWPERTHWAITE, Philip S.; CRAIG, Matthew W.; DEIL, Christoph; GUILLOCHON, James; GUZMAN, Giannina; LIEDTKE, Simon; LIAN LIM, Pey; LOCKHART, Kelly E.; MOMMERT, Michael; MORRIS, Brett M.; NORMAN, Henrik; PARIKH, Madhura; PERSSON, Magnus V.; ROBITAILLE, Thomas P.; SEGOVIA, Juan-Carlos; SINGER, Leo P.; TOLLERUD, Erik J.; DE VAL-BORRO, Miguel; VALTCHANOV, Ivan; WOILLEZ, Julien; ASTROQUERY COLLABORATION; A SUBSET OF ASTROPY COLLABORATION. astroquery: An Astronomical Web-querying Package in Python. 2019, vol. 157, no. 3, p. 98. Available from DOI: 10.3847/1538-3881/aafc33.
18. PYDANTIC. *Pydantic models* [online]. [visited on 2025-11-18]. Available from: https://docs.pydantic.dev/latest/concepts/models/.
19. STIEL, Bjoern. *Worker and the pool* [online]. [visited on 2025-11-18]. Available from: https://celery.school/the-worker-and-the-pool.
20. MDN. *Using HTTP cookies* [online]. [visited on 2025-11-28]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies#security.
21. MDN. *Cross-site request forgery (CSRF)* [online]. [visited on 2025-11-28]. Available from: https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/CSRF.
22. DORFMEISTER, Dominik. *Why You Need React Query* [online]. 2023-05. [visited on 2025-11-21]. Available from: https://tkdodo.eu/blog/why-you-want-react-query.
23. MDN. *WebGL: 2D and 3D graphics for the web* [online]. [visited on 2025-11-22]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.
24. VIS.GL. *Using Layers* [online]. [visited on 2025-11-22]. Available from: https://deck.gl/docs/developer-guide/using-layers.

A Source code

The complete source code is included as an archive attached to the thesis. The complete source code is publicly available in the GitHub repository at `https://github.com/0-mar/AstroCollector`.