

---

# **DATSR System Design Specification**

***Release 1.1***

**Dux D-zine**

**Oct 27, 2022**

## TABLE OF CONTENTS

<b>1</b>	<b>System Overview</b>	<b>1</b>
<b>2</b>	<b>Software Architecture</b>	<b>2</b>
<b>3</b>	<b>Software Modules</b>	<b>5</b>
3.1	Web Interface . . . . .	5
3.1.1	Home Page . . . . .	7
3.1.2	Datasets Page . . . . .	7
3.1.3	High Score Page . . . . .	7
3.1.4	Enter Predictions/Data Page . . . . .	7
3.2	Data Processing . . . . .	7
3.2.1	Data preprocessor . . . . .	9
3.2.2	Data Postprocessor . . . . .	9
3.3	Score Calculator . . . . .	9
3.4	TS Database . . . . .	10
3.5	Database Interpretor . . . . .	13
<b>4</b>	<b>Dynamic Models of Operational Scenarios</b>	<b>16</b>
<b>5</b>	<b>References</b>	<b>20</b>
<b>6</b>	<b>Acknowledgments</b>	<b>21</b>

## **SYSTEM OVERVIEW**

The primary purpose of our application is to serve as a repository made specifically for time series data. Users are able to download training sets of TS data to build and train predictive models. Then, the user can choose to upload their predictions for the data set at which point our application will give them a “score” on the accuracy of their predictions. Our application has the added feature of keeping a “high score” page for predictions made using the available data sets. Furthermore, our repository will allow users to contribute their own data sets that they want to be included in the repo.

The software can be broadly divided into two areas: front-end and back-end. On the front-end side we have a website made with Angular which features [N] user-facing pages. The home page which welcomes users, the data sets page for selecting a time series data set, and a “high score” page for keeping track of the most effective predictive models **[POSSIBLY MORE?]**. All [N] of these pages are made up of a combination of HTML, CSS, Bootstrap, and Javascript which have been generated with Typescript using the Angular web application framework. Also in the front-end, we have Typescript code that handles interactions between the three pages, interfaces with the back-end, and reacts to user inputs.

On the back-end side of things, we have separated the application into two main components: database management and data processing. For the former side of the equation, we have a MongoDB database that interfaces with a Python program (that uses the pandas library) allowing us to request data and contribute new data to the repo. For the data-processing module, we have more Python scripts which handle formatting data going in and out of the database.

## SOFTWARE ARCHITECTURE

The architecture of the software behind our application can broadly be split into three categories: the user interface (i.e. the front-end), the “logic” scripts, and the database/database interpreter.

The users interact with the front-end which primarily uses technologies like Angular, Typescript, HTML, CSS, and Bootstrap. When these user interactions need data processed or calculations done, the work is outsourced to the “logic” section of our program which is primarily implemented in Python with the help of the pandas library. Finally, the database category of modules stores and handles the training data that is available for download on the user-facing site as well as the validation data that will be sent to the “logic” Python scripts.

Therefore, we could visualize a low resolution software architecture overview to be the following.

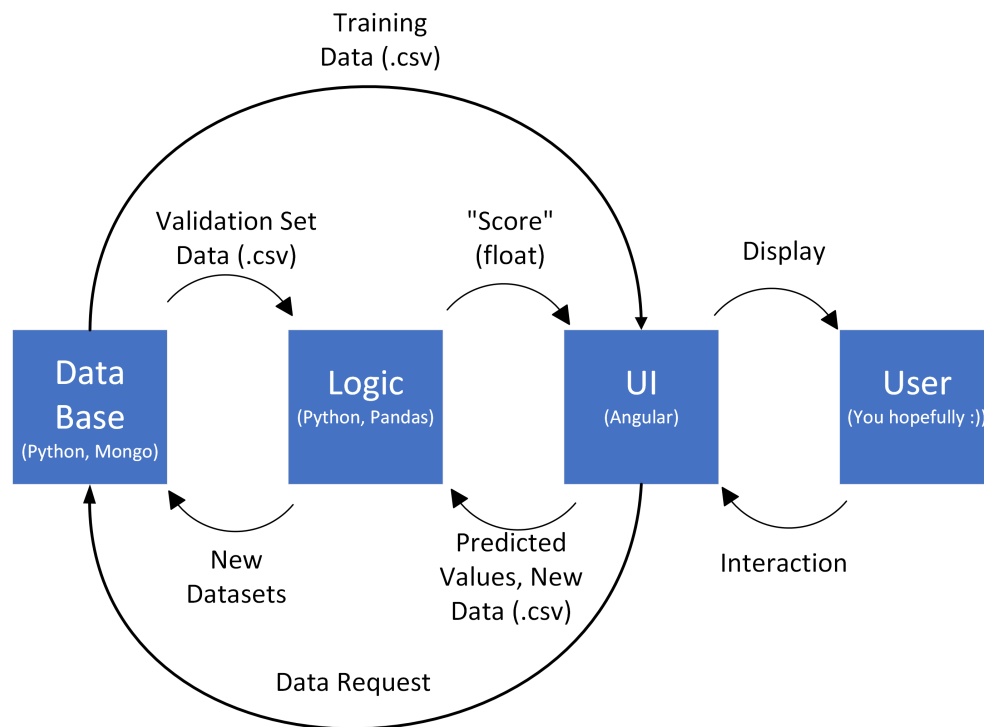


Fig. 2.1: Software Architecture (Low Resolution)

This “category” view of the application may be too broad for some, in which case one can refer to the more detailed “module” view of the application which is pictured below.

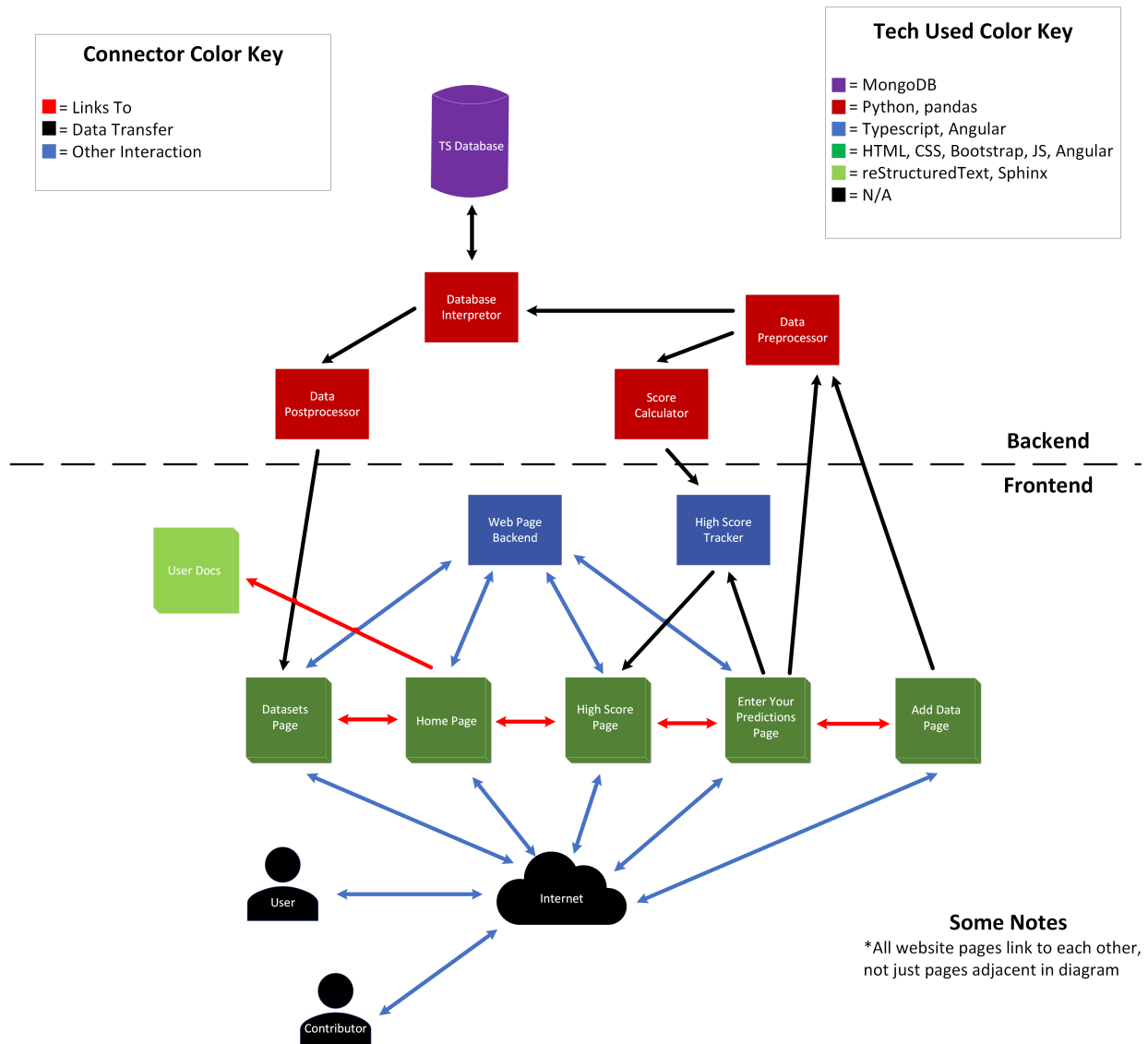


Fig. 2.2: Software Architecture (High Resolution)

In order to elaborate further on the components of that make up this diagram, the following table lists all the modules and sub-modules referenced above, their functionality, and what “category” they fit into in the broader scheme.

Table 2.1: Software Architecture Modules and Sub-Modules

Module/Sub-Module	Category	Functionality
TS Database	Database	Stores time series datasets as well as metadata that describes each dataset.
Database Interpreter	Database	Creates an interface between python and MongoDB which allows for inputting and extracting data from the DB when needed.
Data Preprocessor	Logic	Formats data from contributors so that it can be transferred into the database.
Data Postprocessor	Logic	Formats data taken from the DB so that it can be interpreted by other Python modules and/or downloaded by end-users.
Score Calculator	Logic	Uses statistical error equations to create a “rating” for the effectiveness of a user’s predictions generated for a particular data set.
Home Page	UI	Introduces end-users to the application, gives basic information about the project and the team, and provides links to navigate to other pages in the user interface.
Datasets Page	UI	Displays a list of available time series data sets with descriptions of their key features. Allows users to download .csv files of data sets they wish to use.
Add Data Page	UI	Allows users to contribute to time series data sets to the repository.
Enter Your Predictions Page	UI	Provides an interface so that end-users can get feedback on their predictive models without revealing the validation set of the time series data.

What makes this system operational is not just the individual components, but their interactions with each other. Although the two diagrams above give some idea of the interactions we see in our system, we can elaborate on some key points to further specify the operations of the app.

One key interaction is between the database interpreter module and the data pre-/post- processing modules. These interactions consist of data transfer which allows TS data to pass between the database and the end-users in the correct format. Another interaction that is prominent in the second diagram above [[LINK HERE](#)] is the linking between different pages of the User Interface as well as to the user documentation. This interaction is important because it creates navigability across the application and adds to the intuitive nature of the user interface. Finally, we can look at the key interaction between the backend scoring script and the frontend scoring pages. These interactions consist of data transfers as well as control instructions which allow calculation to be done by the score calculator module and have the user interface react appropriately to the results of those calculations.

The architecture choices that resulted in the system design we see above were made with the primary functionality of the application in mind. We want to effectively create a pipeline for users to access and contribute to our repository of time series data sets. The way our application is set up, users simply have to interact with the intuitive UI in a manner specified in the documentation and the rest is handled by behind-the-scenes modules that process and format data. Furthermore, our decision to implement the system in terms of areas (front-end vs back-end) that are divided into categories (database, logic, UI) which are then divided into modules allows for modifiability and flexibility.

## SOFTWARE MODULES

### 3.1 Web Interface

This module's primary function is to provide the user an interface through which they can interact with our application. Specifically, it allows them to access and add to the TS data in our repository as well as test predictive models and compare to other models' results.

The structure of this module can be visualized with the following static model.

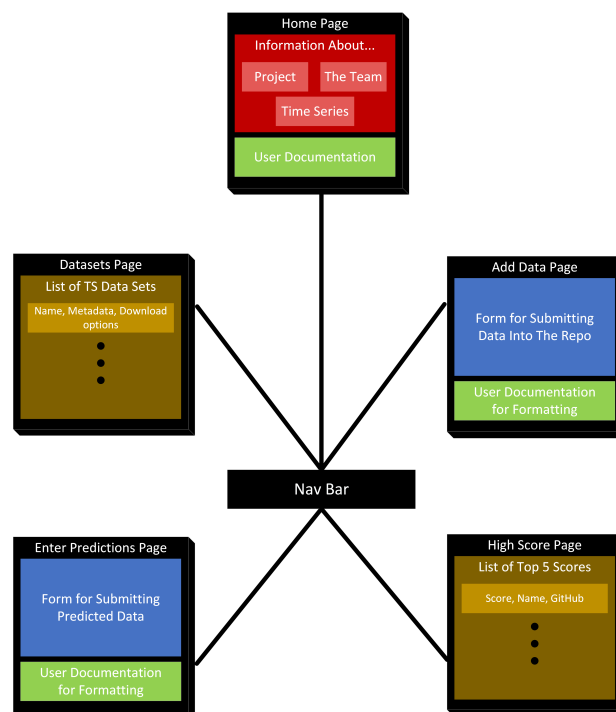


Fig. 3.1: Static Model of Web Interface

This module interacts with the typescript code that makes up the framework that the site is built upon. It controls interactions between the pages, passes on requests to reach any back-end modules, and updates the user-facing pages with generated HTML and CSS code. This module is also the primary part of the application which interfaces with end-users. End-users are able to reach the web interface through connecting to a UOregon server by way of the Internet and from there they can interact with the application using options provided on the web site.

To further show how this module interacts with other modules the following dynamic model further specifies the module's interactions with the other components of the system.

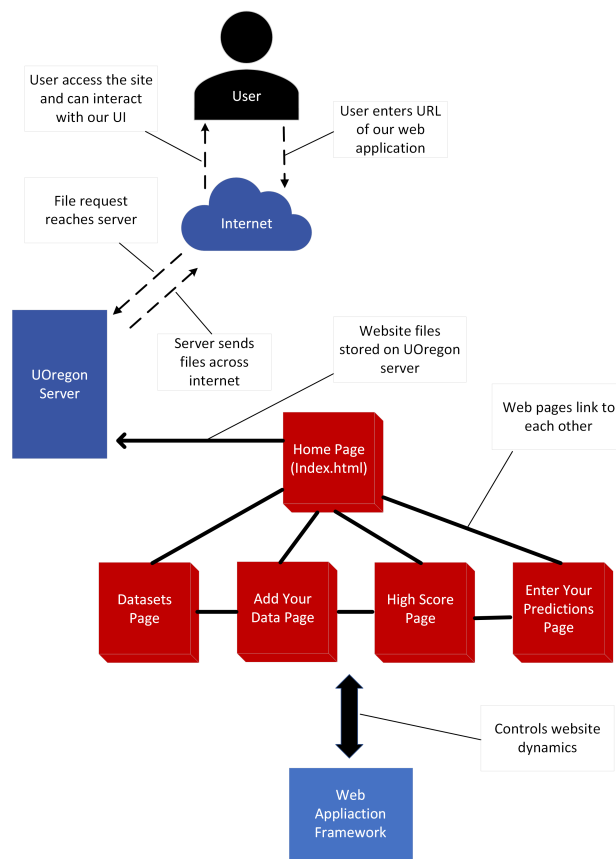


Fig. 3.2: Dynamic Model of Web Interface



The rationale behind the design decisions for the web interface is primarily familiarity. Separate pages with a navigation bar to link them is familiar to most Internet users, making navigation and comprehension intuitive. Our reasoning with the technologies we choose was that our team had familiarity with Angular and Angular provided us with all the functionality we needed.

The reason we favored this design over others is that it provided the most straight forward user interface. More complicated and detailed user interfaces could be created, but with our design users are only provided with what they need. This simplifies user documentation, increases loading speeds, and optimizes efficiency for users.

This module can be divided into five sub-modules which are listed below.

### **3.1.1 Home Page**

The purpose of the home page is to welcome the user to the application and orient them to the website interface. Specifically it allows them to easily navigate to other pages and learn more about the project if they so choose. Furthermore, the home page links directly to user documentation, which is a thorough guide if they have any trouble using the app.

### **3.1.2 Datasets Page**

This page displays a list of time series data sets which are available to download in the form of .csv files. Information about each data set is displayed on this page including the number of variables, the number of data-points, and the domain that the data comes from.

### **3.1.3 High Score Page**

This part of the web interface allows users to see how their predictive methods compare to peers who also use the repository. They are ranked in terms of a “score” which is calculated using an equation specified below.

### **3.1.4 Enter Predictions/Data Page**

Here users can enter predicted values that they have generated for time series data sets taken from the repository and receive feedback from the application. They can also upload new data sets that they wish to be included in the repository in the future.

## **3.2 Data Processing**

This module acts a mediator between modules by formatting their data outputs so that they can be interpreted as inputs for other parts of the system. The structure of this component can be visualized statically as in the diagram shown below.

However the static model above doesn’t do the best job at showing the purpose of this module which is as a middle ground for transporting data throughout the system. The data processing section of the application receives inputs from both the database interpreter module and the web framework module. The former is in the form of database data which it then processes to be in a format accessible to users of the web interface; while the latter does the opposite—it takes database formatted data and makes it readable by the MongoDB database. From this it follows that the data processing module outputs to the same modules it receives inputs from. The dynamic model below shows this situation in a more visual fashion:

The design decision to include this module was a clear one because MongoDB’s BSON data type is not very accessible to the target users of our application and therefore we need to do some interpretation to accept other formats. The

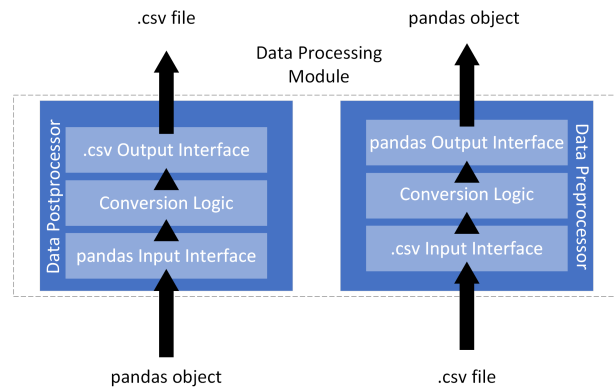


Fig. 3.3: Static Model of Data Processing Module

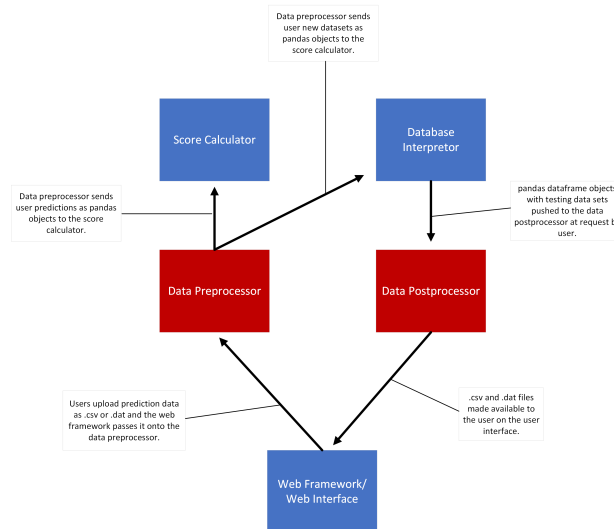


Fig. 3.4: Dynamic Model of Data Processing Module

splitting of this module into two separate components made development much more straightforward as we could focus on a single data translation paradigm (i.e., .csv to BSON or BSON to .csv).

The other design decisions that were considered were primarily in the realm of what data types should be accepted and what should be the “mediator” data type. For the data types accepted, we considered JSON and .txt, but ultimately settled on .csv and .dat as these were generally what other TS repositories had their data available in. The “middle-ground” datatype is a pandas dataframe object which we choose due to its plentiful options and its through documentation. It also fit well with both the Python modules in the backend and the Python interface provided by MongoDB.

The data processing module of our application can be divided into two sub-modules based on the direction in which data is flowing.

### 3.2.1 Data preprocessor

This part of back-end system processes time series data inputted as a .csv file and formats it as a pandas object which is then passed on to the database interpreter module to be formatted to comply with MongoDB’s interface formatting.

### 3.2.2 Data Postprocessor

This sub-module formats data outputted by MongoDB into .csv files which can be passed on to the front-end and made available to users.

## 3.3 Score Calculator

This module calculates the “score” of predicted values submitted by users of the repository to give them an idea of how accurate their predictive model was. It uses the following equation to do its calculation:

[equation here]

The score calculator is a module that is essentially defined by its interaction with other modules and doesn’t have much internal structure of its own. The following static model displays a more static view including some internal structure, but notice that it still includes a good amount of interaction with other modules as this is the score calculator’s primary purpose.

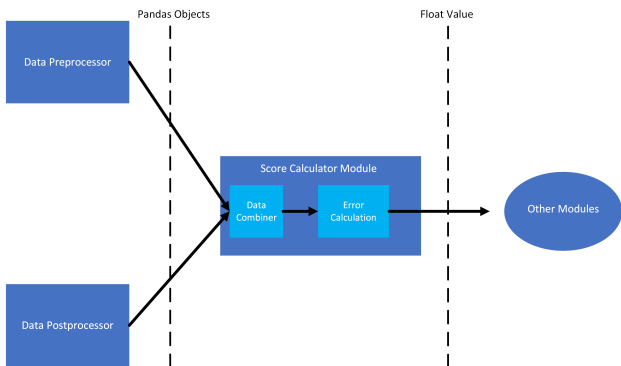


Fig. 3.5: Static Model of Score Calculator Module

To further elaborate on the interface of this module with the rest of the system, we can split the interfacing into two categories: inputs and outputs. The score calculator receives inputs from the data preprocessor and database interpreter. The inputs come in the form of pandas dataframe objects unless they are control messages. The output comes in the form of a float value representing the “score” calculated using the equation above. The modules which receive the score

value are the web framework and the score tracking module (by way of the web framework), which eventually makes its way to the web interface to be displayed properly.

The dynamic model of the score calculator (shown below) visualizes this module's interface in a more clear way:

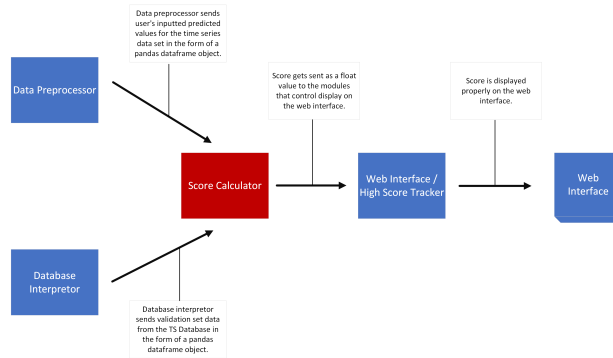


Fig. 3.6: Dynamic Model of Score Calculator Module

The design decisions of this module were made primarily with statistics in mind. The language and libraries used are popular in data science (Python, pandas, numpy) and the equations used in calculation are common error measures across many fields.

Other design decisions were considered; however, we favored this approach because it aligns with the needs of our users. ML engineers use evaluation schemes that are heavily based on classical statistics and so we wanted to comply with industry and research standards when building this module.

## 3.4 TS Database

This module stores the time series data that is offered as the main service of the repository. It also contains meta data about the time series data sets that are stored including the number of variables, the number of data points, and the domain that the data comes from.

Two other collections exist within the database as well: an archive of users who have submitted predictions and a queue of additions to the repository that have not yet been approved by a reviewer in the system. These provide further functionality in the app in the form of score tracking and growing the repository's data set.

We can see a visualization of this compartmentalization of the TS database with the following static model:

The interactions that this module has with the rest of the system is very straightforward due to the existence of the "database interpreter module" whose sole purpose is to interface with the database using procedures defined by MongoDB. The following dynamic model for the TS database module goes into further detail about the interface between the interpreter and database.

This module was implemented using MongoDB, python, and python libraries (primarily pandas). Many of the design decisions around the TS Database were made with these technologies in mind—playing to their strengths and accommodating to the way they interact with other technologies. Furthermore, our design decisions allowed for greater usability for us as developers. This was key as we are a team of somewhat inexperienced programmers.

Many other designs were considered, but the primary alternative we looked at was a relational database such as SQLite. Ultimately, we decided that a non-relational database would be preferable because of its more straightforward interface with Python and its more intuitive data formatting protocols.

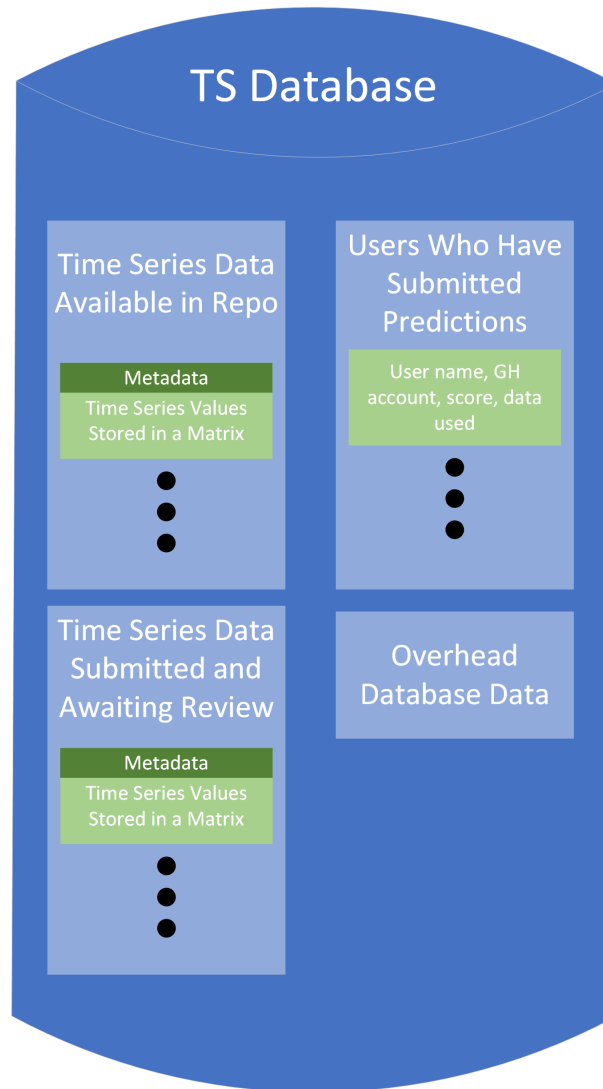


Fig. 3.7: Static Model of TS Database Module

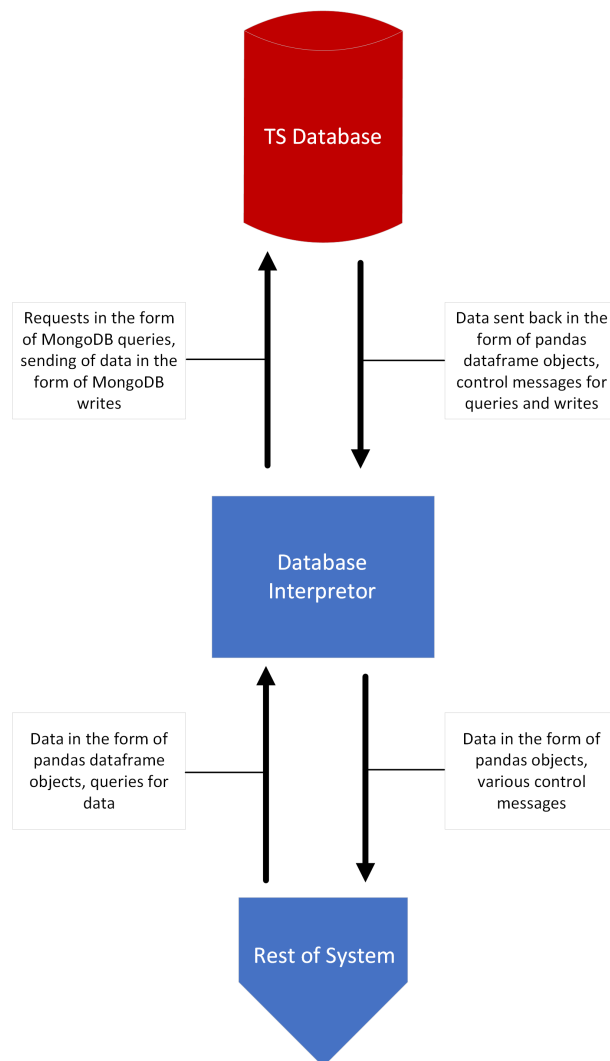


Fig. 3.8: Dynamic Model of TS Database Module

## 3.5 Database Interpreter

The purpose of this module is to query the database when data is needed by users and to write data into the database when contributors want to expand the repository. This module is essential to the system because it allows the TS Database (which provides the primary functionality of the repository) to be incorporated into the application.

Like the score calculator, the database interpreter is essentially defined by its interface with other modules. However, a static model of the module can be used to give a better idea of how the interpreter handles its interactions. The diagram below shows such a model:

The interface that this module has with other modules is defined by the interactions the TS database requires in order to be a functioning part of the application. Because of this, we can think of the MongoDB database as the primary module it interacts with in the form of reads and writes compatible with MongoDB's Python interface.

On the other side of the application, the interpreter receives incoming data transmissions strictly from the data pre-processor (this is because data must be formatted as a pandas object before being moved into the TS database). The database interpreter also receives control message inputs from the web framework when a validation set is needed in order to calculate a user's score.

In terms of outputs, the database interpreter transfers data to the data postprocessor when it needs to be transferred directly to users and to the score calculator when it will be used to create a score value. The interpreter may also send control messages to other modules in order to notify them of receiving or not receiving certain requests.

The design of this module was created with ease of interface in mind. Because the database interpreter acts primarily as a conduit to the TS database storage, we wanted to make this module efficiently interact with MongoDB as well as the modules on the other side of the application.

When designing this module, we considered using a more command line oriented approach that is supported by MongoDB. However, the python interface libraries provided by Mongo were more than enough to create effective passages of information and control messages. Furthermore, by staying in the domain of Python code it was easier to integrate this module with the rest of the backend which was also primarily written in Python.

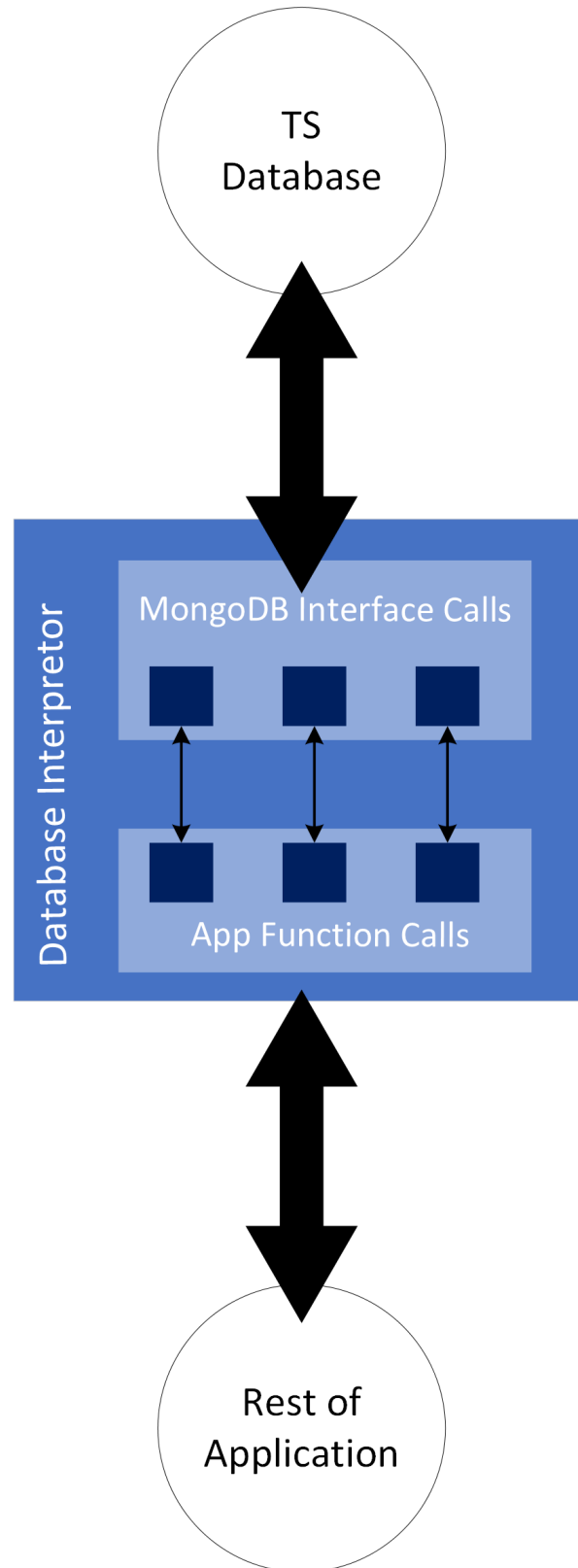


Fig. 3.9: Static Model of Database Interpreter Module



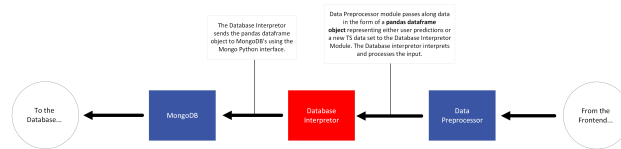


Fig. 3.10: Dynamic Model of Database Interpreter Module

## DYNAMIC MODELS OF OPERATIONAL SCENARIOS

There are three use cases that are specified in the system requirements specification for this system. The design for the operation of each of these use cases is described visually below. For the sake of continuity, the color coding of the individual modules follows the same key as the high resolution software architecture diagram shown above (Fig. 2.2).

The first diagram shows how our application is able to provide users with the ability to download data sets from the web interface.

The diagram below shows how a user can upload their predicted values and receive a score after they have downloaded the training portion of a time series data set.

The next diagram is a visualization of the process in which a contributor can add a time series data set to the repository.

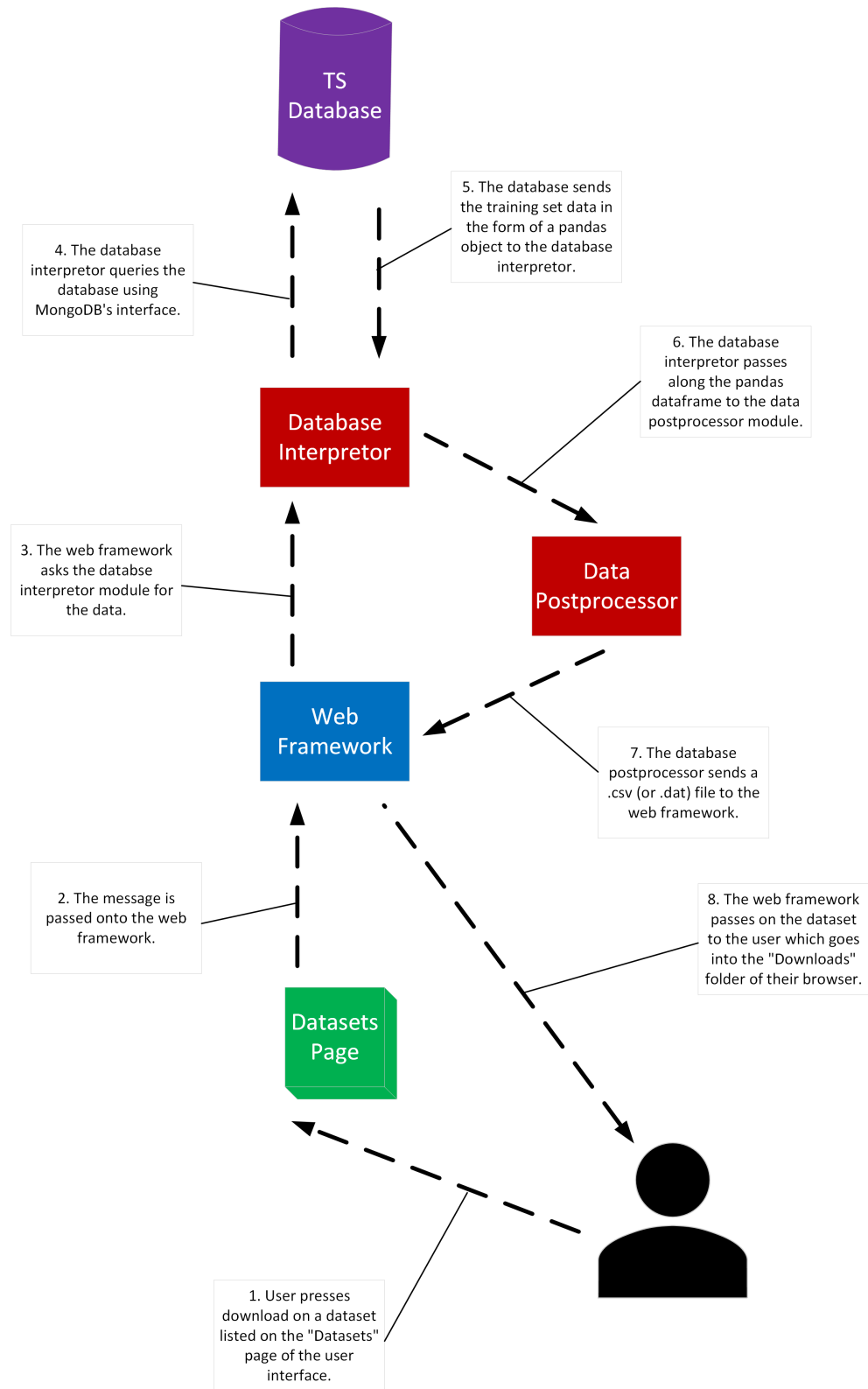


Fig. 4.1: Use Case #1 (Retrieving Time Series Data)

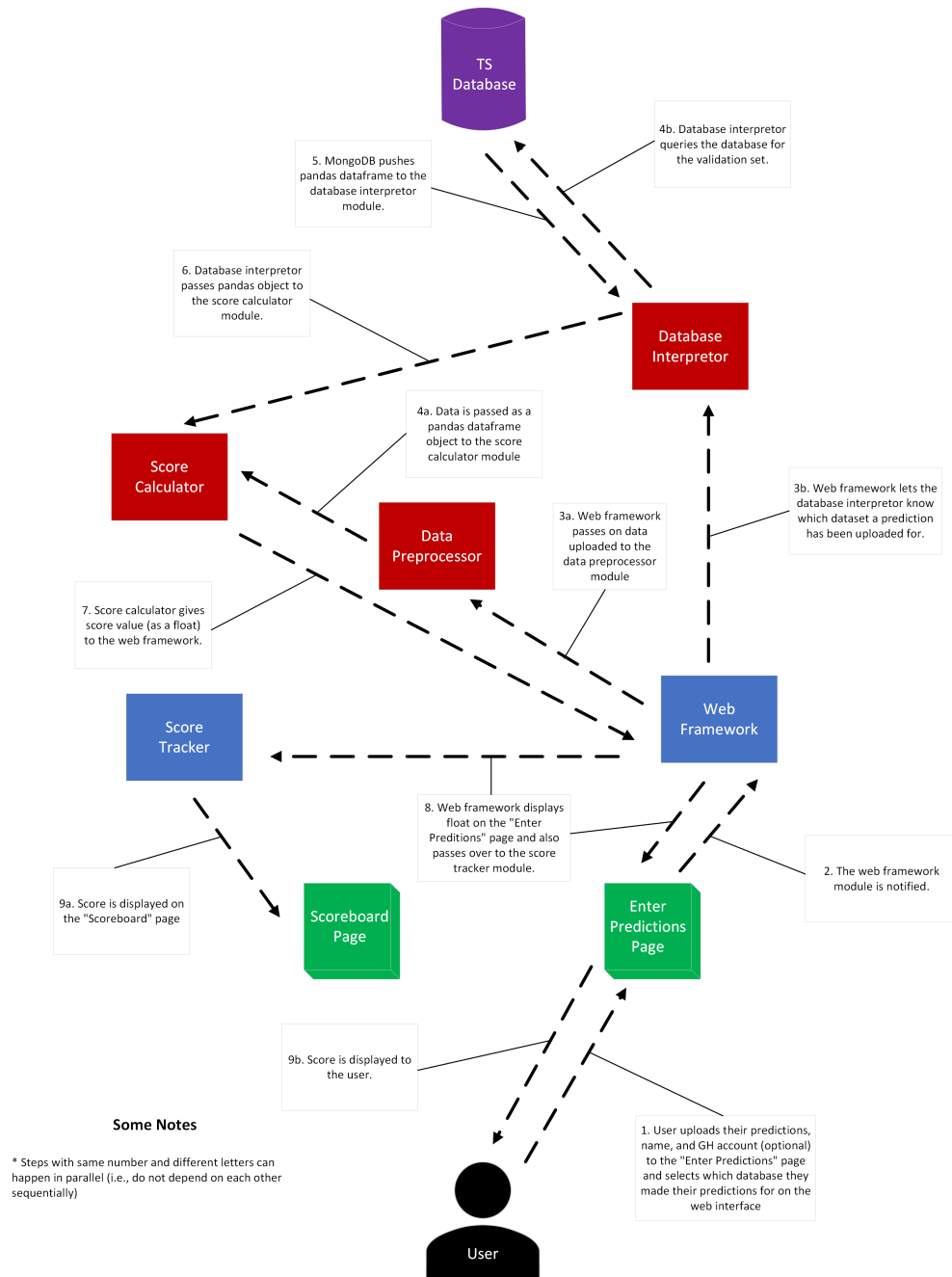


Fig. 4.2: Use Case #2 (Uploading Predictions)

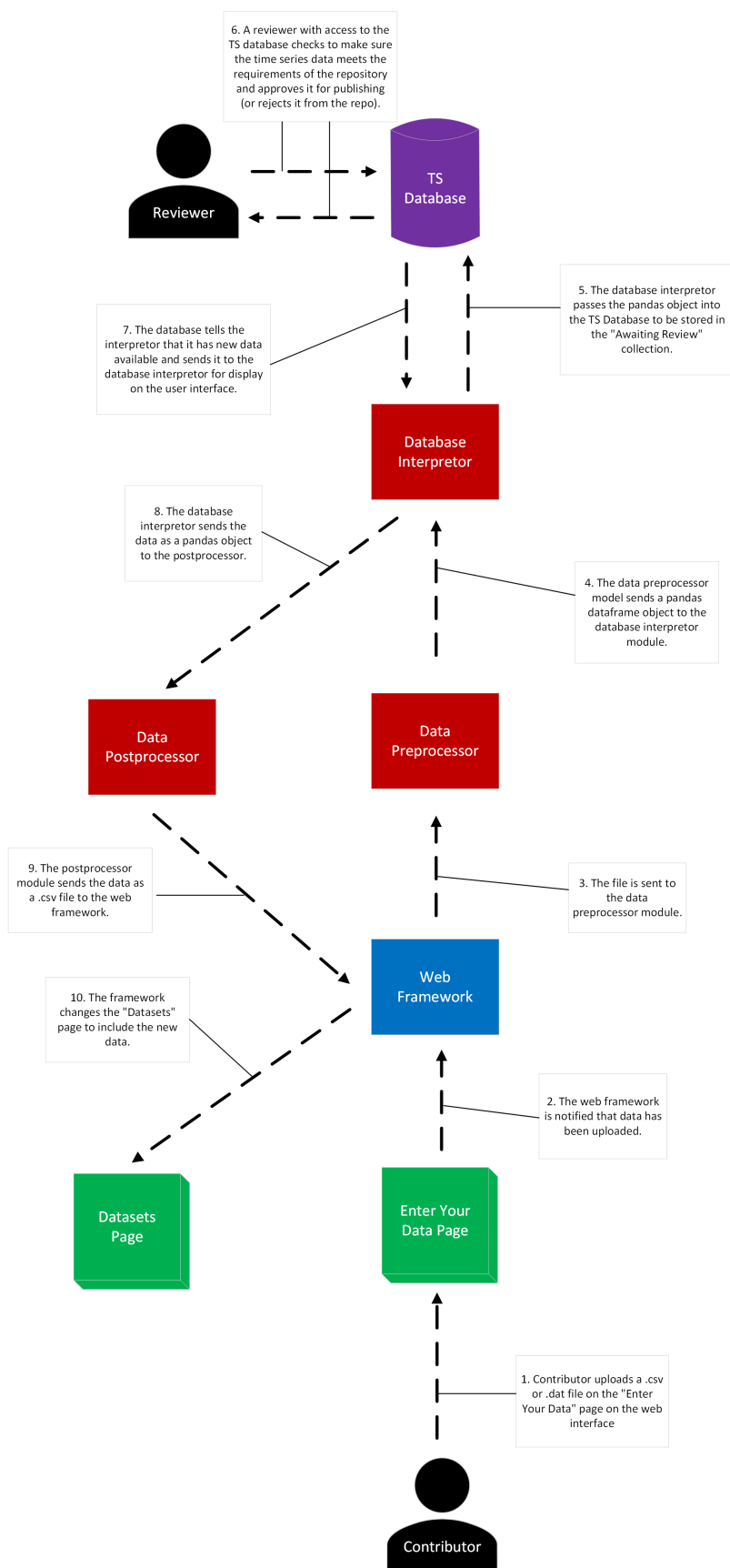


Fig. 4.3: Use Case #3 (Adding Data to the Repository)

**REFERENCES**

## **ACKNOWLEDGMENTS**

The format of this SRS document was originally created by Professor Juan Flores. Reference to a completed SRS document was provided by Ronny Fuentes, Kyra Novitzky, Jack Sanders, Stephanie Schofield, Callista West with their Fetch project SRS.