DATSR System Design Specification

Release 1.2

Dux D-zine

Oct 30, 2022

TABLE OF CONTENTS

CHAPTER

ONE

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 for each data set. Our repository also allows contributers to add data to the collection.

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 four user-facing pages. The home page which welcomes users, the data sets page for browsing time data sets, a page for uploading data, and a "high score" page for keeping track of the most effective predictive models. All four 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 pages, interfaces with the back-end, and adjusts 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 system, 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 functions which handle the formatting of data going in and out of the database.

CHAPTER

TWO

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 interpretor.

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 using libraries such as pandas and numpy. Finally, the database section of our modules stores and handles the time series data for the application.

Therefore, we could visualize the software architecture in lower resolution using the diagram below (Fig. 2.1).

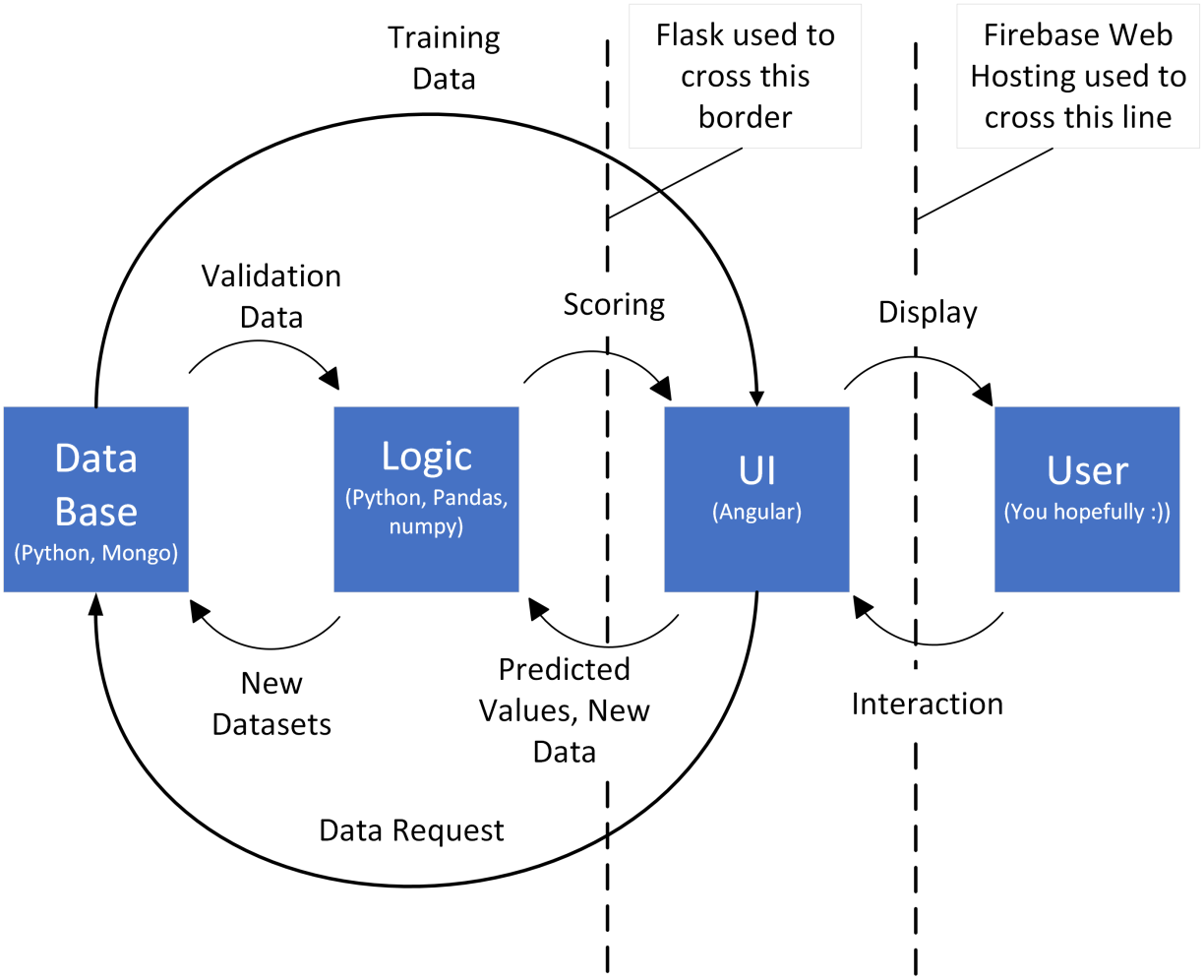

Fig. 2.1: Software Architecture (Low Resolution)

This "category" view of the application may be too broad for some, in which case one can refer to the more detailed modular view of the application below (Fig. 2.2).

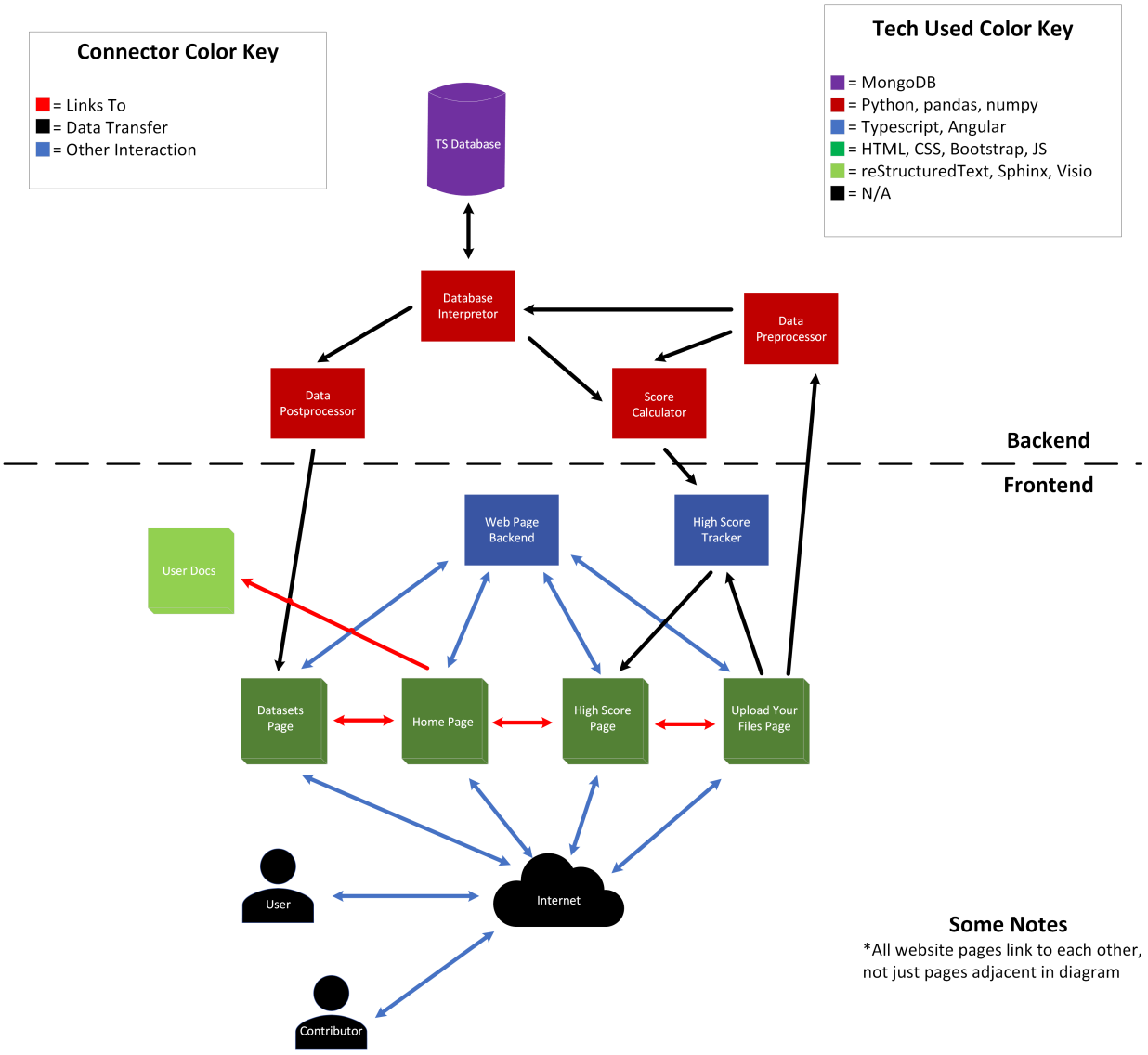

Fig. 2.2: Software Architecture (High Resolution)

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 them.
Database Interpretor	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 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.
Upload Your Files Page	UI	Allows users to contribute to time series data sets to the repository and provides an interface so that end-users can get feedback on their predictive models without revealing the validation set of the time series data.
High Scores Page	UI	Shows users how other predictive schemes have performed on the data provided in the repository.
User Docs	Documentation	Guides users through using the application.

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 elaborate on our design.

One key interaction is between the database interpretor 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 (Fig. 2.2) 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. Our mission was 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, format, and move around the data. Our decision to hierarchically implement the system in terms of areas (front-end vs back-end), categories (database, logic, UI), and modules allows for modifiability and flexibility.

CHAPTER

THREE

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 results with other users.

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

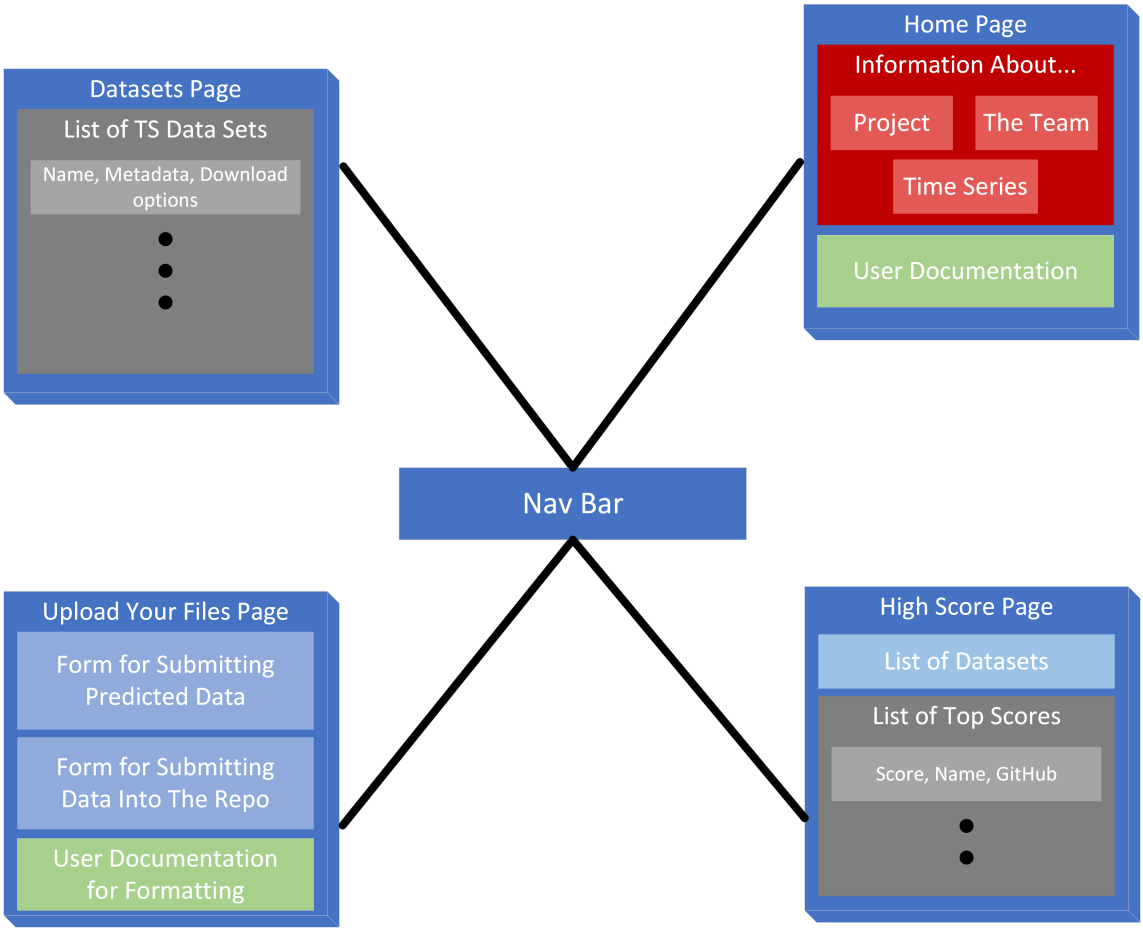

Fig. 3.1: Static Model of Web Interface

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 component of the application that interfaces with end-users. End-users are able to reach the web interface through any browser and from there they can interact with the application using options provided on the web UI.

To further show how this module interacts with other parts of the system, we can look to the following dynamic model (Fig. 3.2):

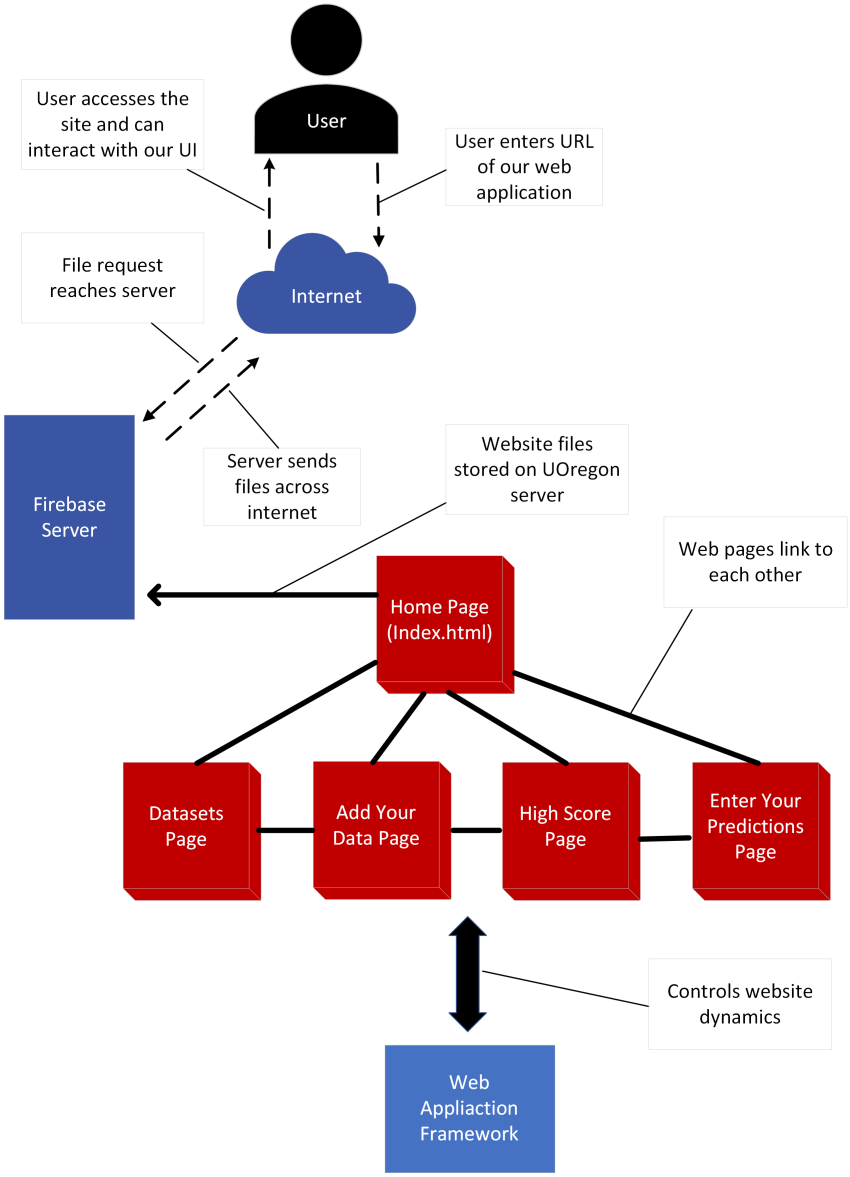

Fig. 3.2: Dynamic Model of Web Interface

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 intricate user interfaces could be made with other technologies, 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 to 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 or .dat 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. Users can see a high score list for each individual data set. They are ranked in terms of a "score" which is calculated using an equation specified in the "Score Calculator" section below (Section 3.3).

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 in the diagram shown below.

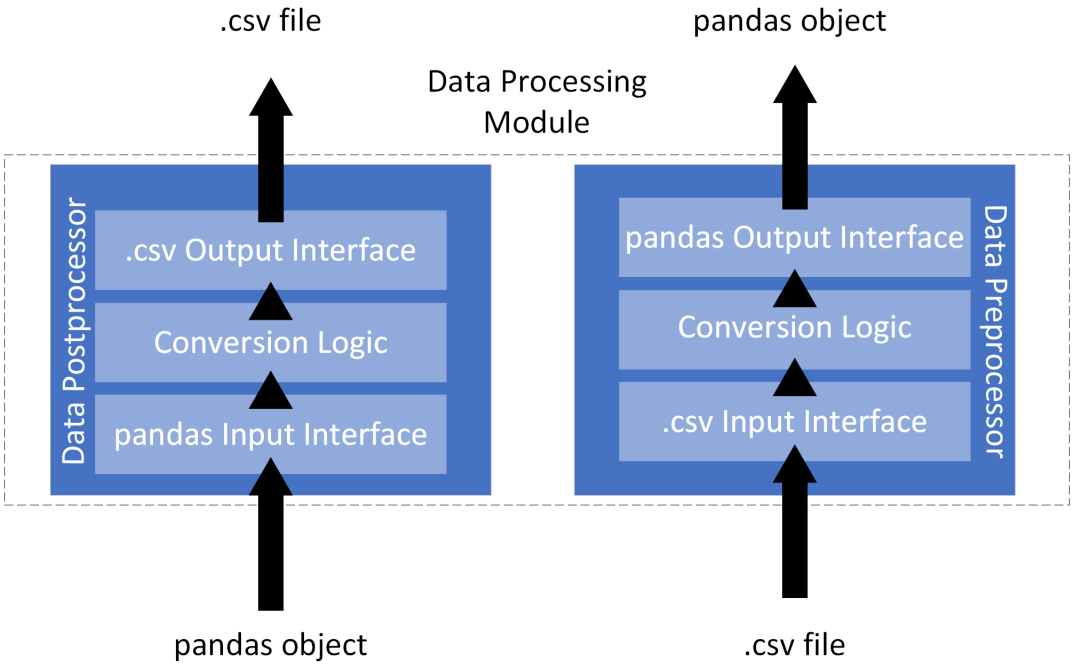

Fig. 3.3: Static Model of Data Processing Module

However the static model above doesn't do the best job at showing the purpose of this module as a middle ground for transporting data throughout the system. The data processing section of the application receives inputs from both the database interpretor 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 is the opposite–it takes user formatted data and makes it readable by the MongoDB database. As a "mediator," 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:

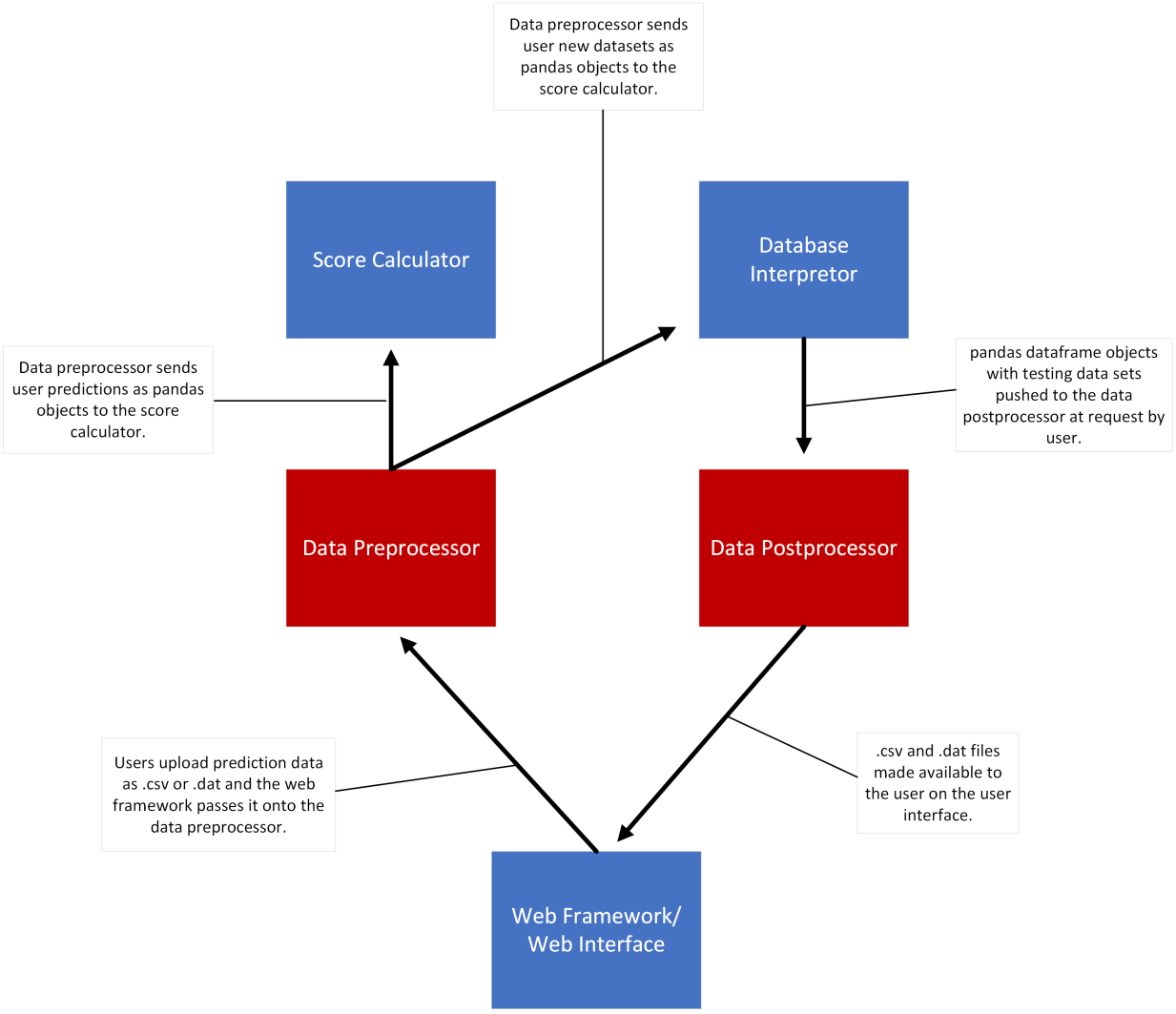

Fig. 3.4: Dynamic Model of Data Processing Module

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, so accepting other data types was essential. The 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/.dat to BSON or BSON to .csv/.dat).

Other design options that were considered were primarily in the realm of what data types should be accepted and what the "mediator" data type should be. For the data types accepted, we considered JSON and .txt, but ultimately settled on .csv and .dat. These are generally what other TS repositories had their data available in and both are common in many fields of AI/CS research. The "middle-ground" datatype is a pandas dataframe object which we choose due to its plentiful options and thorough documentation. It worked very well with all the Python modules in the backend.

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/.dat file and formats it as a pandas object. The pandas dataframe object is then passed on to the database interpretor so that it can be modified to comply with Mongo's python interface.

3.2.2 Data Postprocessor

This sub-module formats data outputted by MongoDB into .csv/.dat files which are given to the front-end modules and made available to the 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 following static model displays one view of this component that focuses on its internal structure.

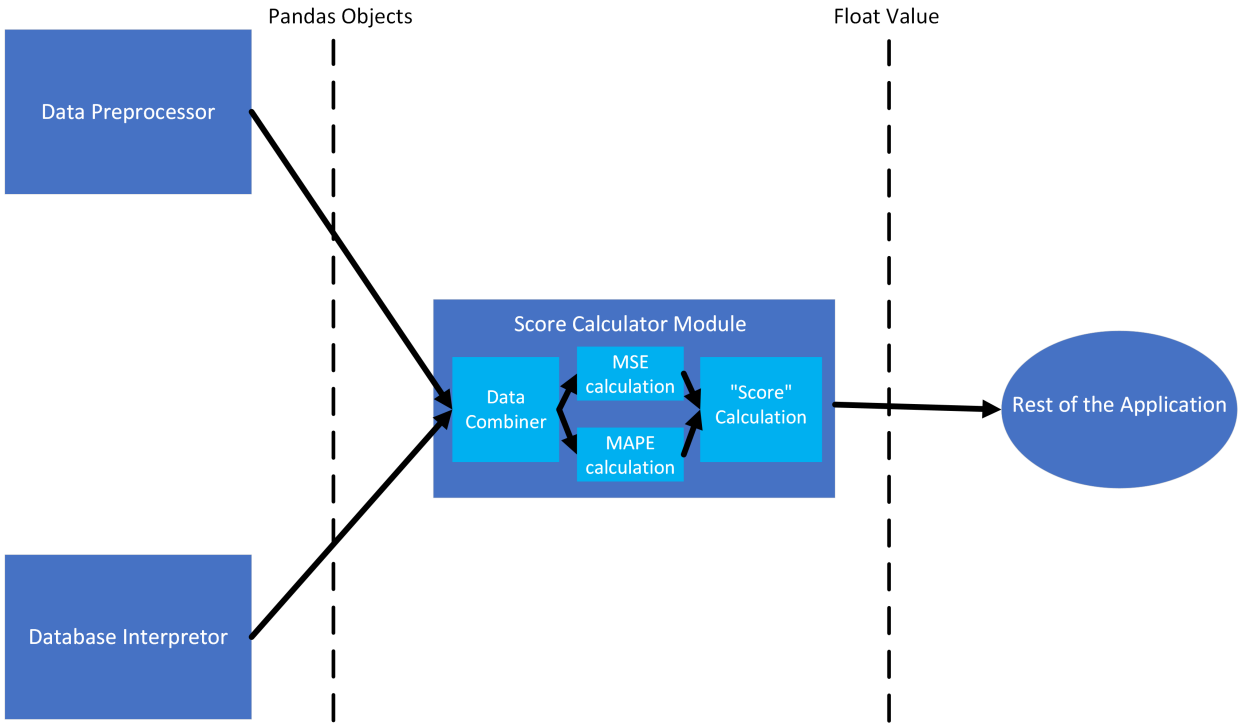

Fig. 3.5: Static Model of Score Calculator Module

To further elaborate on the interface of this module with the rest of the system, we can split the interactions into two categories: inputs and outputs. The score calculator receives inputs from the data preprocessor and database interpretor. The inputs come in the form of pandas dataframe objects. The output comes in the form of a float values representing the "score" calculated using the equation above as well as the individual statistical values that were compiled to generate the "score." The modules which receive the score value are the web framework and the score tracking module, which eventually makes its way to the web interface to be displayed properly.

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

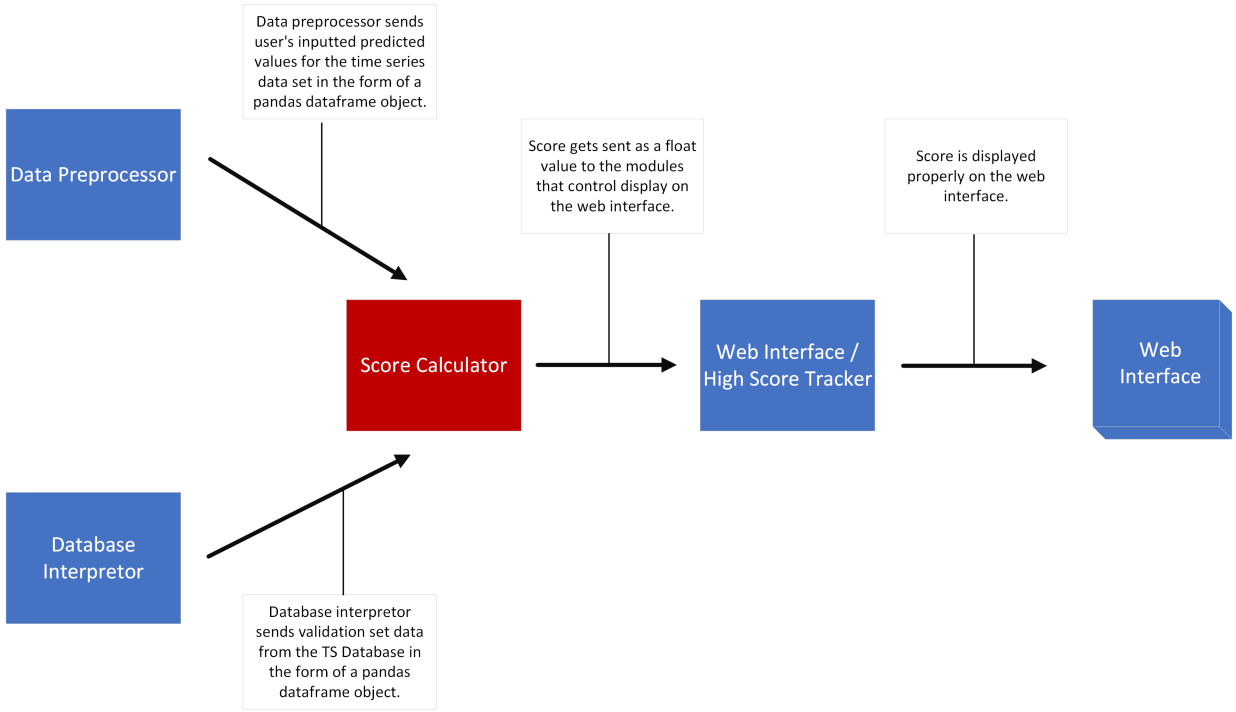

Fig. 3.6: Dynamic Model of Score Calculator Module

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 and academic researchers 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 data sets that are stored including the number of variables, the number of data points, and the domain that the data comes from.

Three other collections exist within the database as well: an archive of users who have submitted predictions, the scores of each dataset, 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 including tracking scores and growing the repository's data set.

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

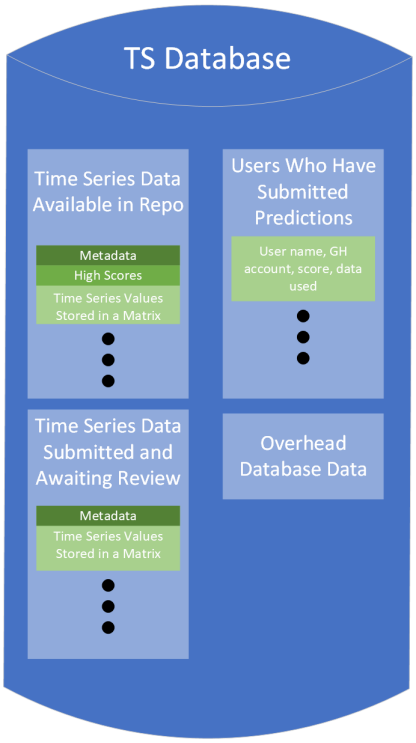

Fig. 3.7: Static Model of TS Database Module

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

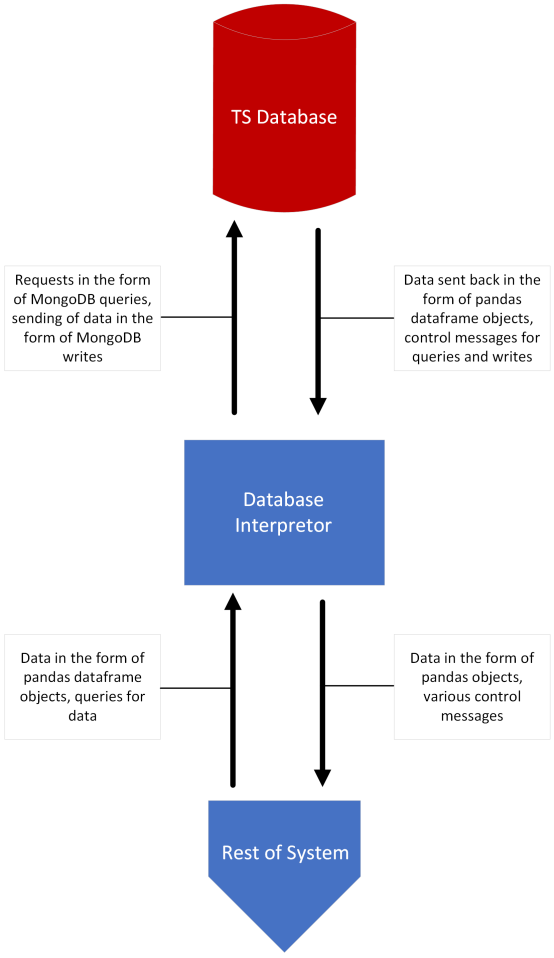

Fig. 3.8: Dynamic Model of TS Database Module

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.

3.5 Database Interpretor

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 smoothly into the application.

The database interpretor 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 interpretor is structured. The diagram below shows such a model:

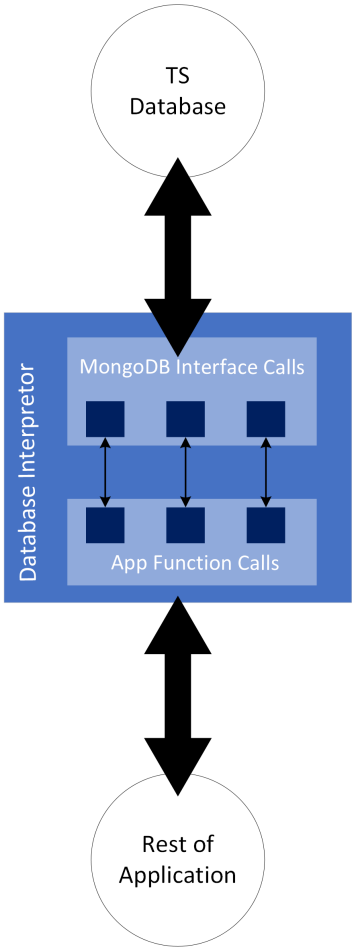

Fig. 3.9: Static Model of Database Interpretor Module

The interface that this module has with other modules is defined by the interactions the TS database requires in 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 interpretor receives incoming data transmissions from the data preprocessor (this is because data must be formatted as a pandas object before being moved into the TS database). In addition to TS data, the database interpretor is given updates for the scoreboard from the Web framework. The database interpretor 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 interpretor 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 interpretor also sends information directly to the frontend framework in some cases–specifically for the high scores page.

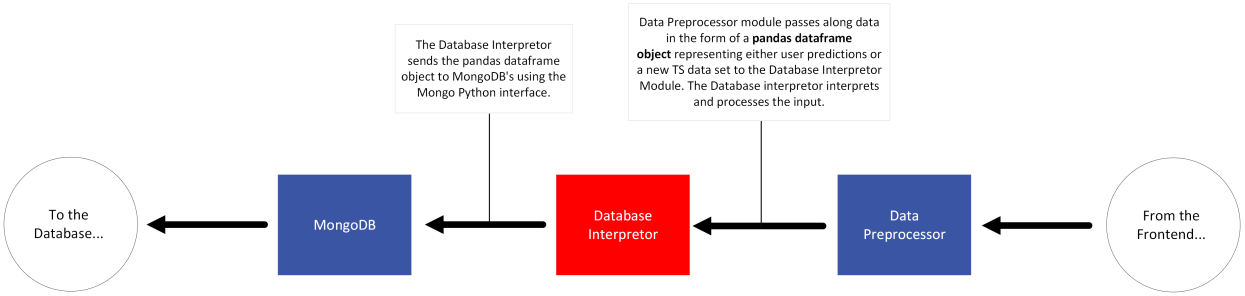

Fig. 3.10: Dynamic Model of Database Interpretor Module

The design of this module was created with ease of interface in mind. Because the database interpretor 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.

CHAPTER

FOUR

DYNAMIC MODELS OF OPERATIONAL SCENARIOS

There are three use cases that are detailed in the system requirements specification for DATSR. 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 (Fig. 4.1) shows how our application is able to provide users with the ability to download data sets from the web interface.

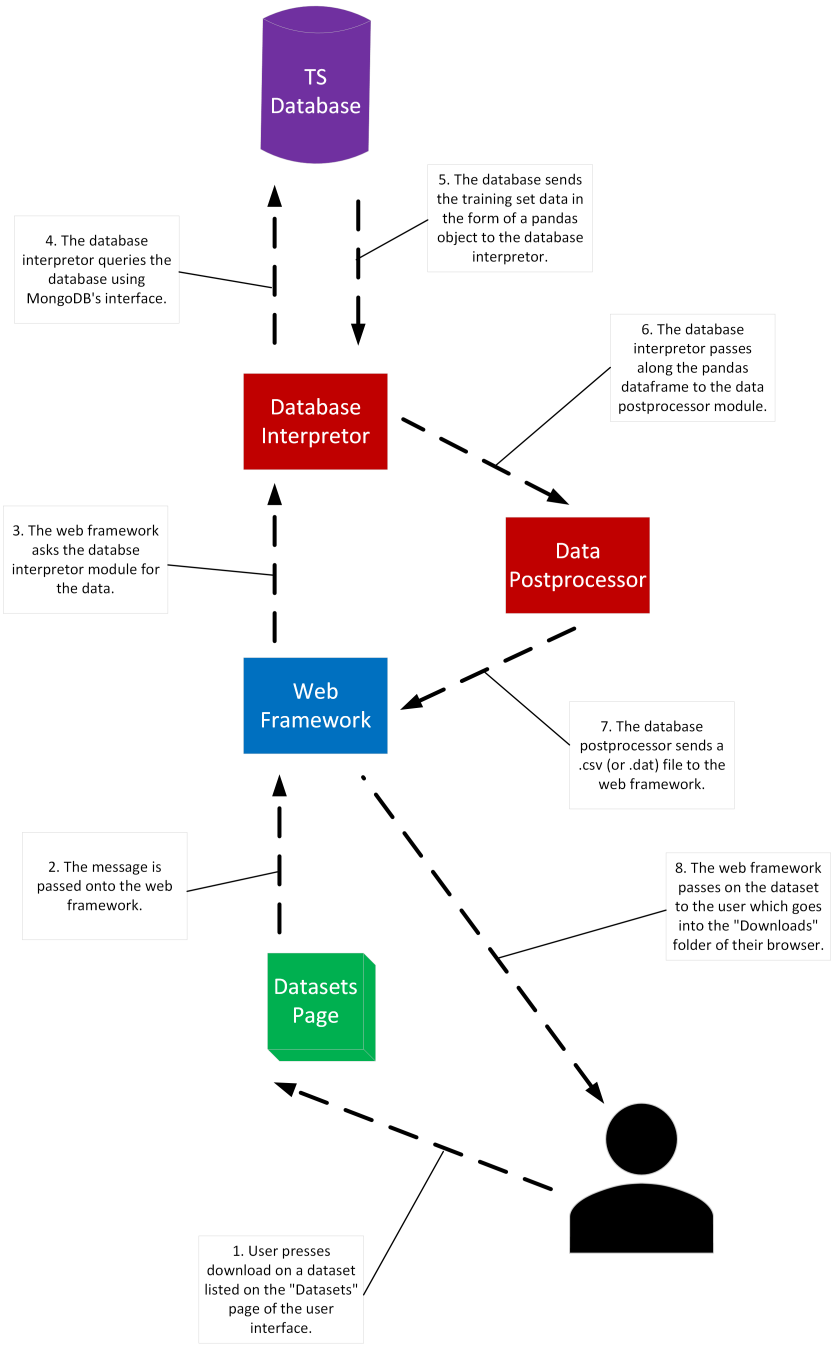

Fig. 4.1: Use Case #1 (Retrieving Time Series Data)

The diagram below (Fig. 4.2) 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.

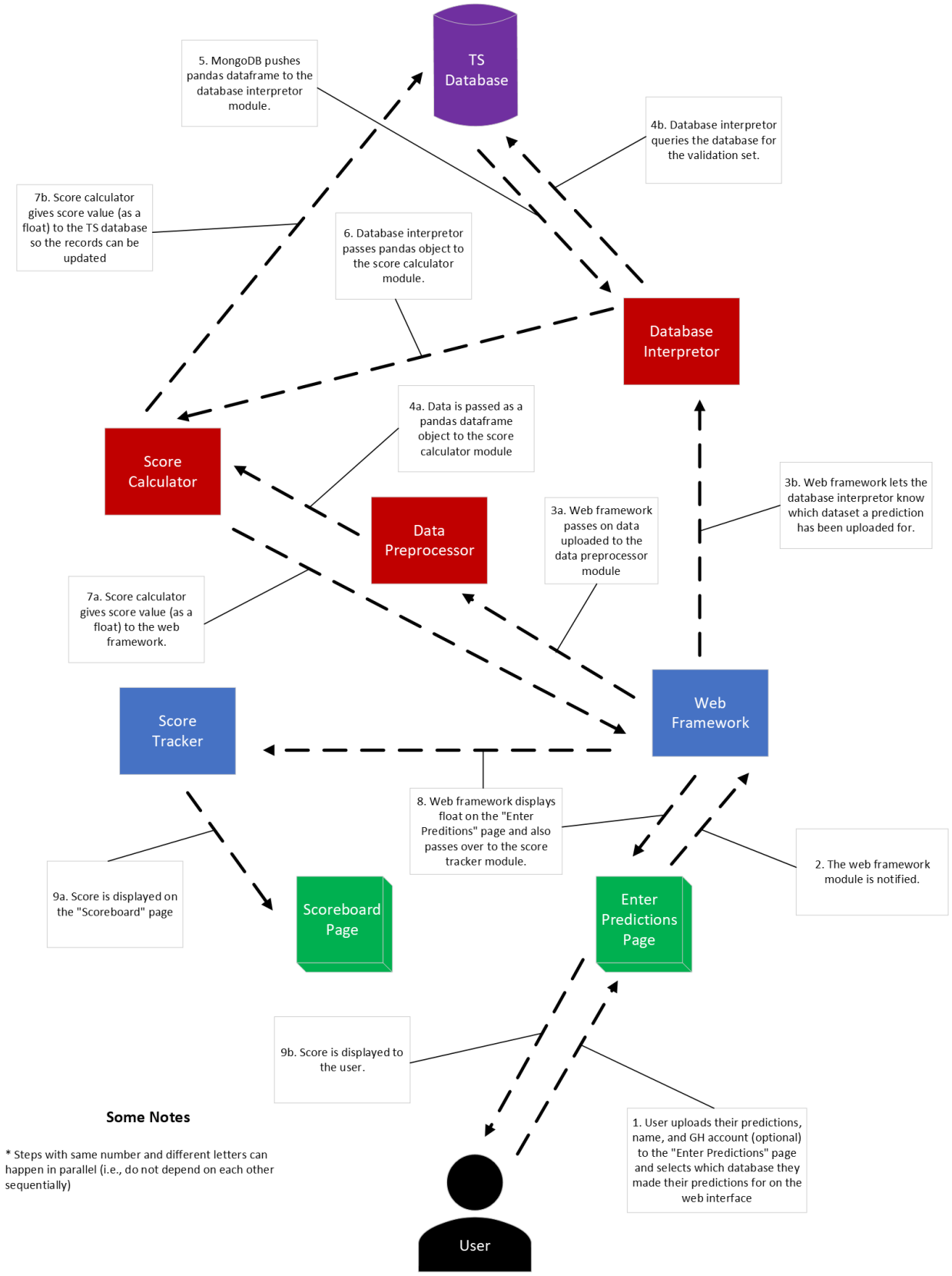

Fig. 4.2: Use Case #2 (Uploading Predictions)

The next diagram (Fig. 4.3) is a visualization of the process in which a contributor can add a time series data set to the repository.

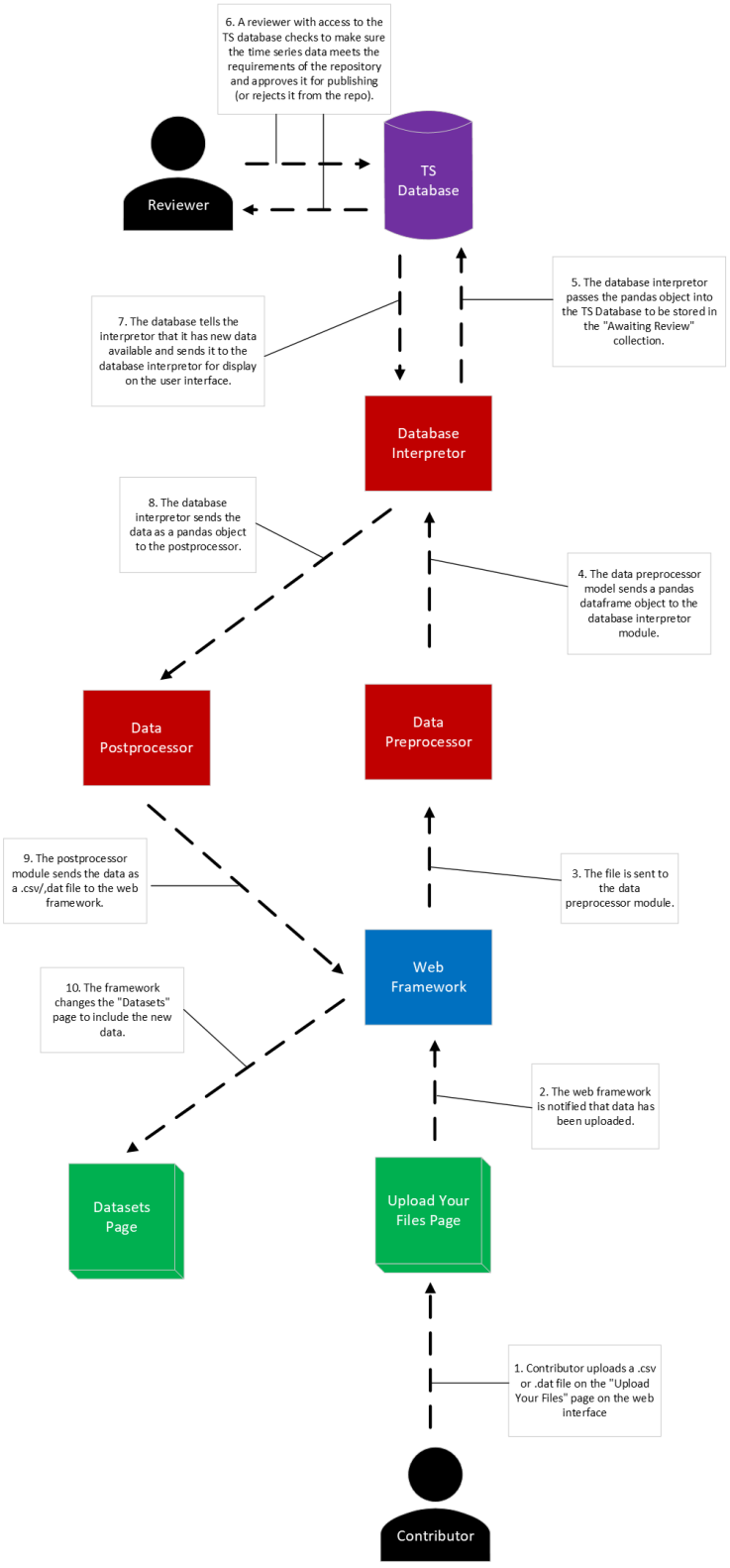

Fig. 4.3: Use Case #3 (Adding Data to the Repository)

CHAPTER

FIVE

ACKNOWLEDGMENTS

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