CECS 491A - Section %

December 13th, 2018

High Level Design Document

[Team Spyderz]

[Product: CheckIt]

Jonathan Asencio [Project Leader] - Back End - 014245983

Alex Philayvanh - Back End - 017508814

Bryan Bare - Back End - 011741260

Kunal Patel - Front End - 013329054

Table of Contents

Introduction

This high level design document for our single page application (SPA), CheckIt, will go over the high level decisions that will help create this website. This design document will also explore the general flow of activities on the website such as registering a user or adding items to a watchlist. Diagrams are provided to demonstrate the logical (abstract) view and physical (network) view of our system. In addition, this document will discuss the stack of technologies used on the front end and back end for our application. This SPA will be built with a model view controller(MVC) architectural design on the front end, and ASP.NET Web Api for the back end. More details containing the benefits of using these architecture patterns for an SPA will be discussed further in the document.

The logical view of our system will include an abstractions diagram which helps show the components that will interact with each other for the system. The physical view will show literal pathways of data and requests through the different hardwares. Our physical stack includes a front end stack and a back end stack. The front end will be done with a CSS Framework, JS Framework, and an AJAX Library. The back end will use the recommended Microsoft Stack. These two stacks will help us develop and be ready for deployment on the AWS infrastructure.

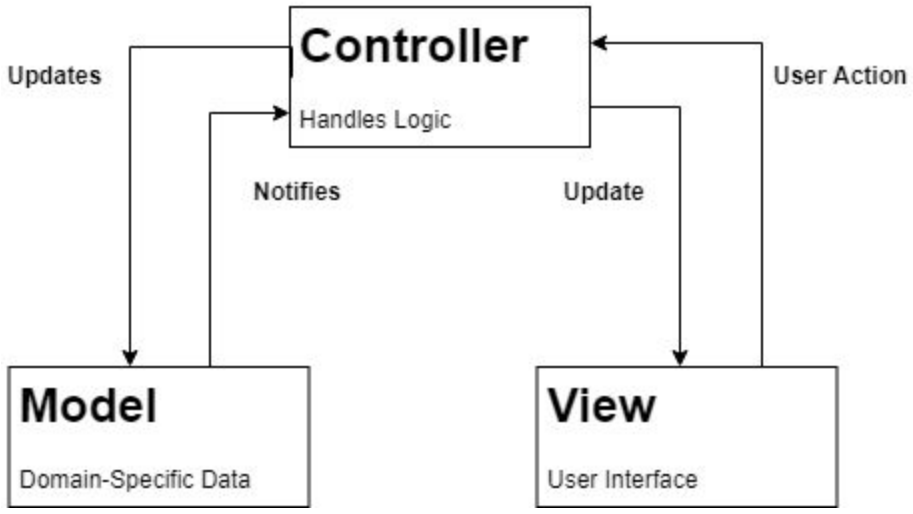
Logical View

Our system will be a Single-page Application. What this means is that instead of loading new pages on every request from the browser, the SPA will dynamically rewrite the current page. This allows the system to flow smoother and act almost as a desktop application would. Data within the system is transferred as JSON using AJAX communication. These formats help provide a smooth transition between requests without the need to reload the page.

When entering the system for the first time, information flows from the data store to the UI to be displayed. This would be the only time the flow starts from the backend without a specific request.

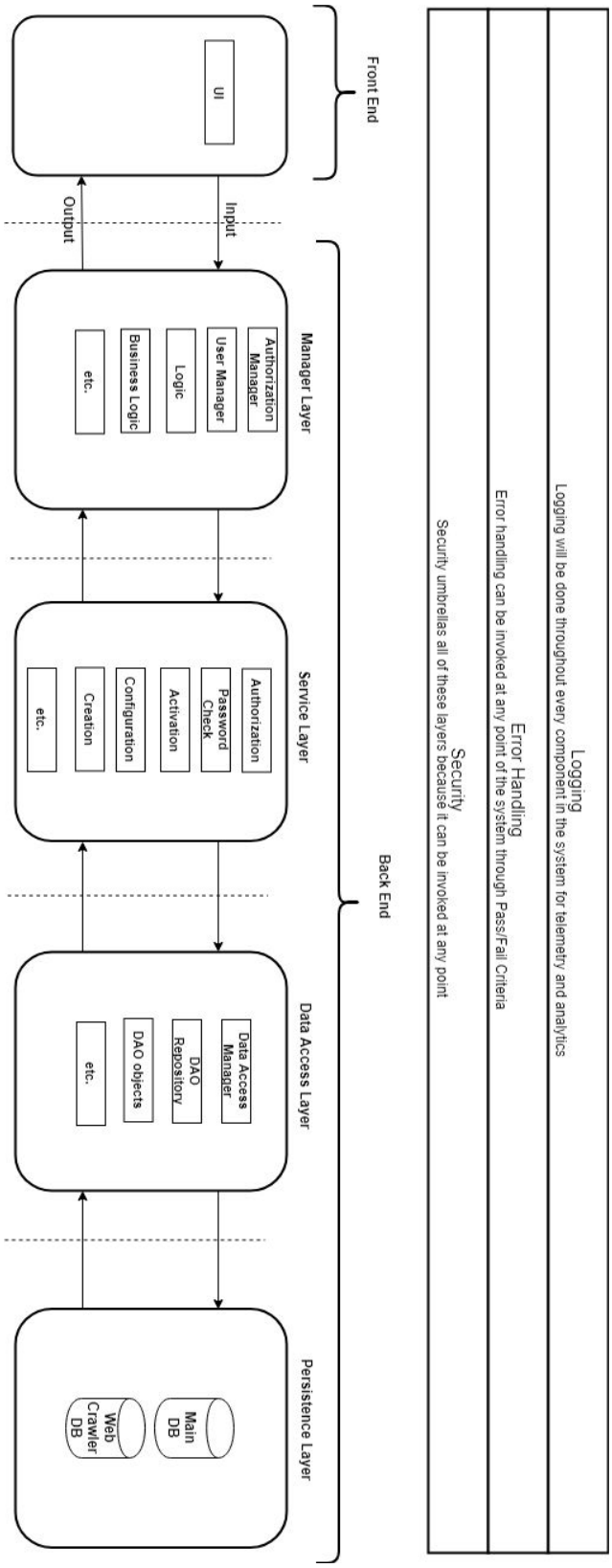
To design the front end, we decided to adopt the MVC architectural pattern to help organize everything. Through this pattern we were able to map out the logical flow easier; and from that, the specific abstractions.

MVC architectural pattern:



- The View will handle most of our front end and UI
    - Data between View and Controller is simply a request and response relationship. The view receives a request to show a certain thing, and the controller handles how to make the response
- The Controller will act as the middle layer which handles all of the transitions from our data store to the UI. Controller will be in charge of interpreting information from the model to be ready to send to the view.
  The Controller also will be manipulating information that the model will have, so the information exchange will go both ways.
- The Model will maintain the data for our application. It will talk to the Persistence Layer to retrieve information relevant to any model we need.

Abstraction Layers:



- UI
    - The UI will handle all of our views and be the main source of input to our system. Input includes any mouse actions, text, and other user generated information.
- Manager Layer
    - The entry point into our backend system is the Manager Layer. This includes several event handlers which will handle all incoming requests and then forward accordingly to each component after checking their logic and business logic.
        - Business Logic
            - This layer encapsulates various tasks such as back end validation. This will cross check the current request with our business rules and determine the appropriate path or response.
- Service Layer
    - The service layer calls individual classes that deal with actions within our system. For example, Authorization is a service that assures an action request can be performed.
- Data Access Layer
    - Once a request is made and a call to the database is requires, a service will call a specific data access class to make a query for the required information. The specific DALs will format the data from the database and return a clean response to the service that called it.
- Persistence Layer
    - Finally, the Data Store is accessed and the pertinent information is collected to be transferred back to the view. Our data store will include a relational database and a secondary storage for indexing from our web crawler.

- Information
    - Information travels throughout our backend as objects, and between our front end to back end as a JSON. Specifically, the authorization, authentication and login will use a token.

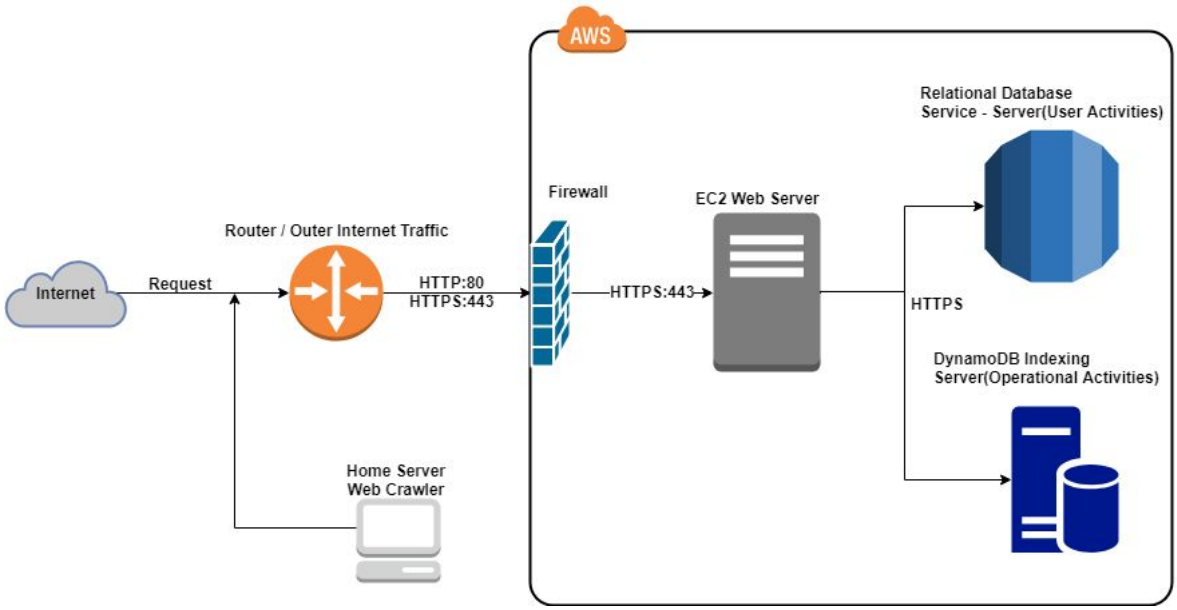
Certain entries to the system will skip certain layers. For example, for our small business promotion, should a business need to upload many items from their business, we would have a way of doing that that doesn't involve loading the full UI. Tasks like these would free up loads of processing time if they don't include the UI. Another scenario is guest users skipping authentication due to not having an account, however they will still be limited in their accessibility to the system.

Physical View

The physical view of our system includes the network diagram. The network diagram will show exactly how data and requests will move throughout our physical system. Since we are using an Infrastructure as a Service (IaaS), the majority of physical system hardware will be remote through AWS. External requests will be directed at AWS along with communication from our home server for the web crawler.

Our firewall will accept both HTTP (port 80) and HTTPS (port 443) requests. However, it will be be configured to elevate HTTP port 80 requests to HTTPS port 443. For better security practices, the EC2 web server will communicate with two separate databases; one for operational activities (which deals with the web crawler), and the other for user activities (which deals with the rest of the system). These two databases will not share any data so replicated entries won't be an issue. This means that they will not communicate directly with each other. Our system will also not require a load balancer because a majority of the system usage will be done by the backend. The system will not need to handle more than 10,000 concurrent users.

Network Diagram:



High Level Decisions

Several things are important in order to design a system. These high level decisions help form the base of the rest of our decisions and will facilitate the design of features on a lower level. Here are the main decisions we made:

- Data will move between the front end and back end in a JSON format
- Errors that occur will be handled through an error handler class and be called when exceptions are found
- Logging will have 3 types of logs: Error log, Telemetry log, and Request log
    - These three will have different columns to easily show distinction and will later help us when creating the dashboard
- We will have our Firewall elevate each request to HTTPS to secure each connection and ensure normality in our entry point.

Dependencies

When designing our system, we realized that some tasks must absolutely be done before others. These are called dependencies. Some dependencies that will help shape the timeline of our system include:

- A Database is the grand dependency
    - It is required for being able to have entities to use within our system
- A Data Access Layer is needed next in order for the rest of the system to access the data of our users
- Logging will be necessary throughout the system in all of the features in order to keep a record of everything that happens.
- Authentication, Authorization and Login will be necessary to access most of our system

- Our search feature that uses the web crawler will be needed to get a majority of our website content. Several things rely on this such as Alerts and Deals of the Day
- Other dependencies are shown by our Project Plan by order of completion

Contingency

While thinking about our project, our group definitely hopes that we can complete each and every task. As we learned quickly, there is a high learning curve for everything and things might not work out the way we plan. As a group, should the need arise, we have decided to remove Business Promotions. This ties in with having business accounts, but we are fairly certain we can at least implement a business side. Simply removing the Promotions sections will allow us more time for other, more important features.