Online Marketplace

Devin Hopkins

Department of Computer Science

Honors Research Project

Submitted to

The Williams Honors College
The University of Akron

Approved:		Accepted:	
Yingcai Xiao	Date: 4/21/2021	[illegible]	Date: 4/21/2021
Honors Project Sponsor (signed)		Honors Department Advisor (signed)	
Yingcai Xiao		Zhong-Hui Duan	
Honors Project Sponsor (printed)		Honors Department Advisor (printed)	
[illegible]	Date: 4/21/2021	[illegible]	Date:
Reader (signed)		Department Chair (signed)	
ZHong-Hui Duan			
Reader (printed)		Department Chair (printed)	
En Cheng	Date: 4/21/2021		
Reader (signed)			
En Cheng			
Reader (printed)			

[illegible]

Date: 4/22/2021

Reader (signed)

Michael L. Collard

Reader (printed)

ONLINE MARKETPLACE

Devin Hopkins

University of Akron

Table of Contents

ABSTRACT

The goal of this project was to create a website that helps a typical person search for items from online marketplaces, such as eBay, Amazon, or Facebook Marketplace. The intention was to ease the burden of needing to search several different websites for the same product. Currently, if a person wants to look for a specific item on multiple marketplaces, they must go to each marketplace individually and search for it. They must enter their specifications repeatedly and load virtually the same web page multiple times. This project's goal was to condense that so the user would only have to put in that information once and then could search all of the marketplaces at once. To create this project, I taught myself the setup of Ruby on Rails and Angular, as well as how to use public APIs. The project is built on the Rails framework with Ruby as the backend language, Angular as the frontend version of JavaScript, a Postgres database, and JSON data objects to transmit data between the front and backend.

INTRODUCTION

OVERVIEW

Online shopping can be a very involved activity when looking for a popular item that sells fast. This is further complicated by the fact that there are so many websites to search. Having an easy way to search all of these sites at once would make for a convenient and efficient way to purchase items. Surprisingly, there are no sites that currently do this. So, I decided to make a website that a user could enter search criteria once, and it would search all online marketplace sites for them. This would not only provide something valuable for everyone but would also help me continue to expand what I have learned in my undergraduate classes and internship.

GOALS

My main goals with this project were to create an application everyone can use and to better understand the inner workings of Ruby on Rails, as well as both Rails' and public APIs. As a side goal, I also wanted an application that had the potential to last into the future. I didn’t want to do something that would be useful for a short period of time and then forgotten about after this project. I chose these goals because I particularly enjoyed using Ruby on Rails during my internship and thought it was a powerful tool that could be used to create something useful. I also didn’t want to do something too easy, and so not only did I create my application from scratch, but my goals also made me learn about public APIs. Before this project, I had never researched public APIs, so I knew I was starting from the beginning with them. I decided to call my project the

Online Marketplace. It’s a simple name that gets across what I’m creating. The fundamental version of the idea is displayed below (Figure 1):

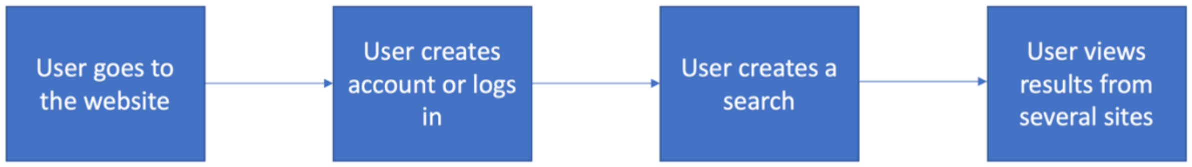

User goes to the website
User creates account or logs in
User creates a search
User views results from several sites

Figure 1

CONCEPTS

As previously stated, the concept for this project was to create an online marketplace for users to come and shop at. The marketplace would pull from several different marketplaces. The user would start by creating an account. Once they created an account, they could then create their own searches. The searches would include keywords, minimum and maximum prices, and any other filters that are commonly present in searches. Other unique filters, such as eBay’s “buy it now” filter, could also be selected. My application would then search all sites the user wanted to search and combine the most relevant search results from each site into one list. The user could then easily browse all of the returned search results in one location. This creates an easy-to-use application that’s also highly convenient.

Purchasing items is also an easy operation for the user. To purchase an item, the user would simply go to its listing on the respective marketplace site by clicking a link next to the item on my site. This also provides the added benefit of not needing to store any sensitive information, such as credit cards, on my site.

ARCHITECTURE

Before getting into the specific design of the application, it's important to know the basic architecture that this application will be using. The Online Marketplace is modeled after a four-tier web application. The typical architecture of a four-tier web application is shown below (Figure 2):

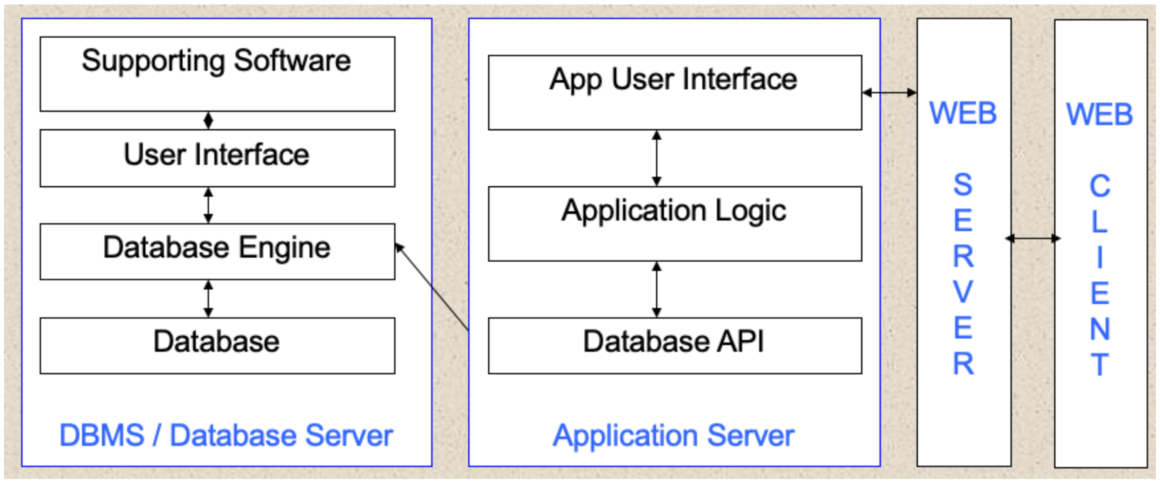


Figure 2

Since I am using Ruby on Rails as my framework, the following page shows the specifics of how the four-tier web application will look with Rails (Figure 3). Note that "RoR" stands for Ruby on Rails.

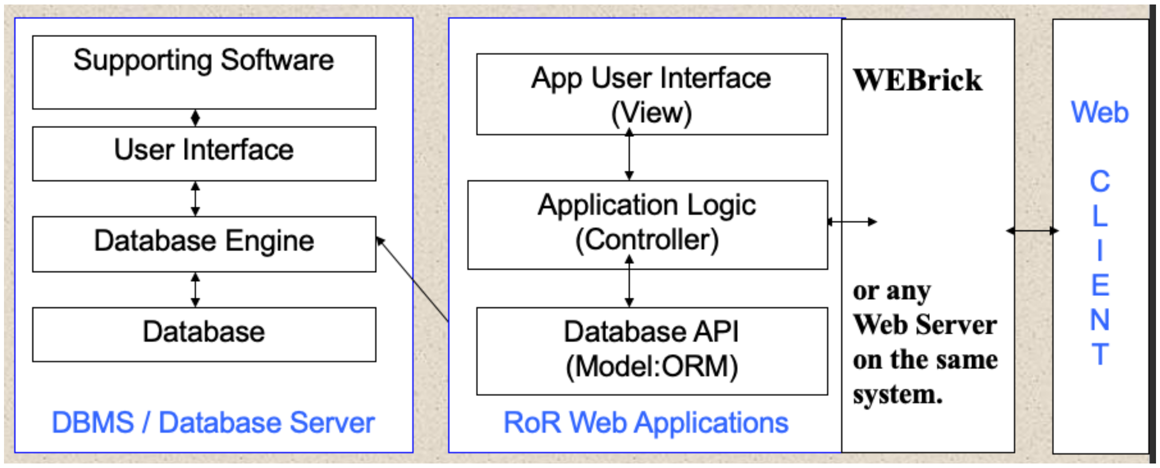


Figure 3

Notice that the Ruby on Rails application uses an MVC design. MVC stands for Model-View-Controller and describes how each part of Rails (the frontend, backend, and database) communicates. I will go into more detail on this later, but for now, Figure 4 shows the basic data flow between each component.

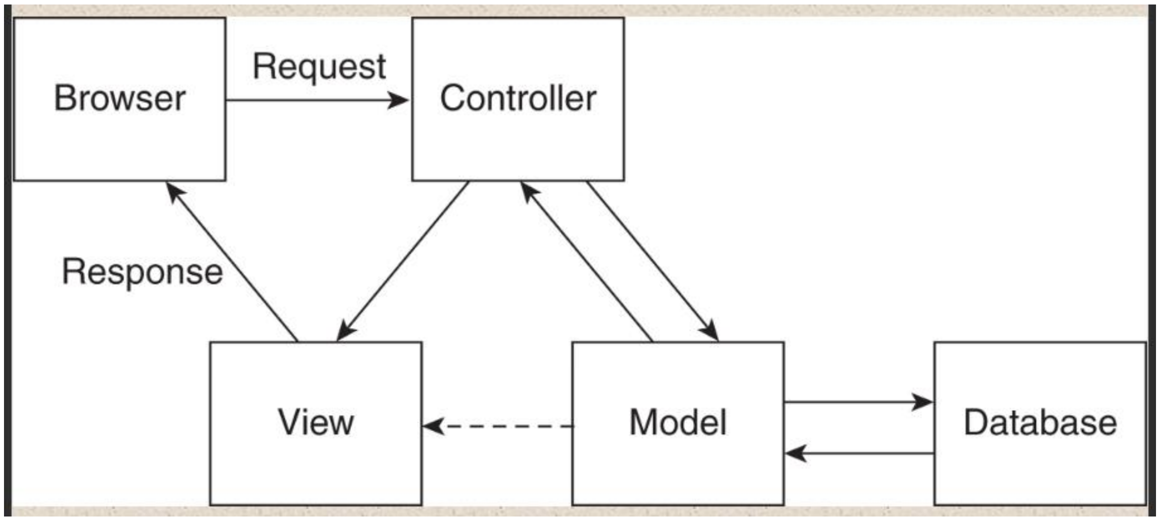


Figure 4

DESIGN

The goal was to have a sleek design that was easy for a first-time user to follow and understand. Below shows the layout of the website (Figure 5 on the following page):

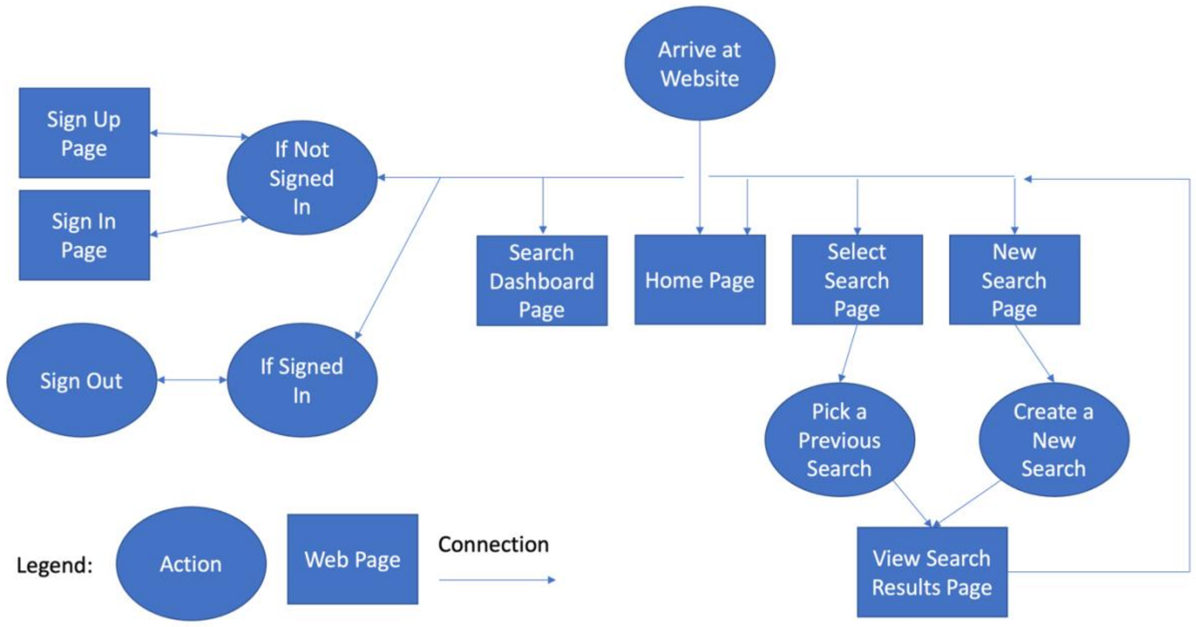


Figure 5

As shown, the user enters the website on the Home Page. If the user is not signed in, they can choose to go to the Sign In or Sign Up Page to sign in or sign up, respectively. If the user is signed in, they can choose to sign out. Regardless of whether or not the user is signed in, they can select to go to the Select Search and New Search Page. After selecting a previous search or creating a new search, the results are brought up on the View Search Results Page. From this page, the user can go to any page.

Each page is designed to serve a specific purpose. The name of each page was designed to indicate what that page's purpose is. The Home Page is designed to be the summary page. When a user goes to the Home Page, they will see a combination of recent searches, search results, and new improvements I've recently made. Unfortunately, I don't have the Home Page's functionality fully implemented yet, and thus it's currently a blank screen. However, pictured below is the navigation bar's look when the user is on the Home Page (Figure 6) and is not signed in compared to when the user is signed in (Figure 7):

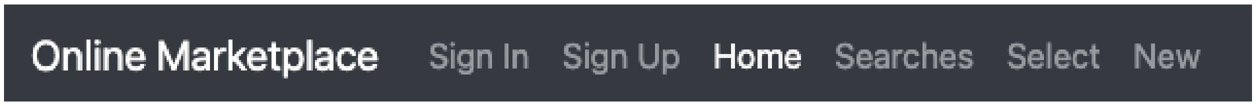


Figure 6

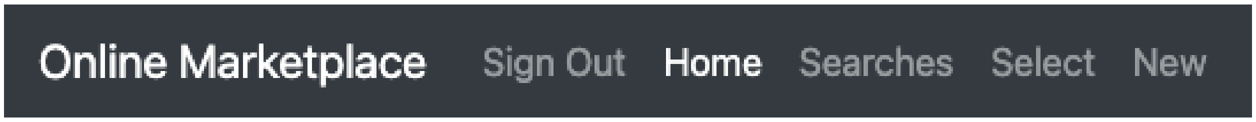


Figure 7

The Sign In and Sign Up Pages are pretty self-explanatory. If the user wants to sign in, they simply go to the Sign In Page and enter their username and password. If they are new and would like to create an account, they simply go to the Sign Up Page and enter all of the necessary information. While the Sign Out option may look like a page, it's not. It's just a button the user can press to sign themselves out. On the following page is a picture of the Sign In (Figure 8) and Sign Up Page (Figure 9 on the following page):

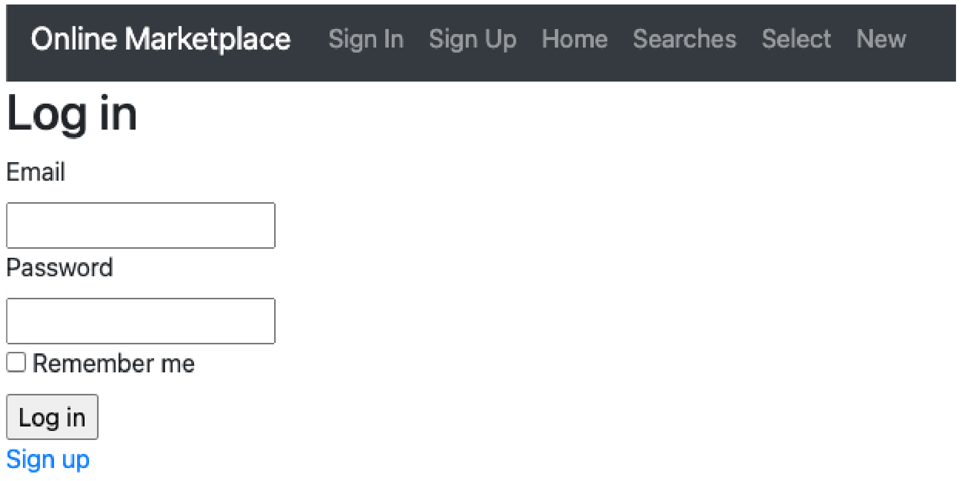
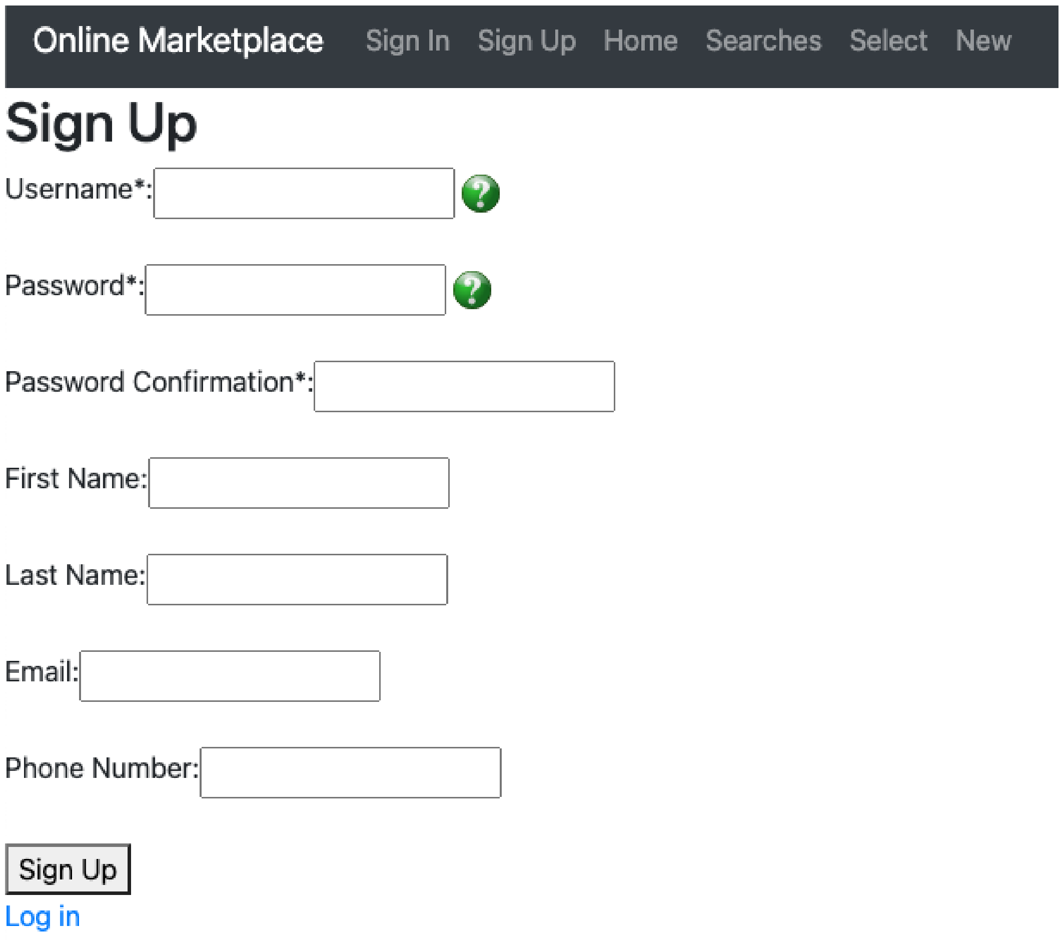


Figure 8



Sign Up

Username*:

Password*:

Password Confirmation*:

First Name:

Last Name:

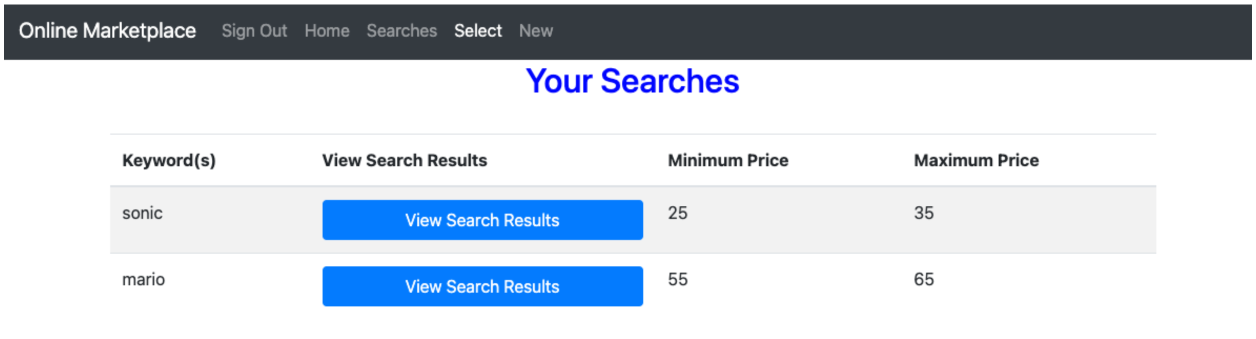
Email:

Phone Number:

Figure 9

The Select Search and New Search Pages are also self-explanatory. If the user has already entered a search and would like to see the search results, they would go to the Select Search Page. This page would list all of their searches out, allowing them to select whichever one they wanted. Once they selected a search, they will be taken to the View Results Page to view the results of the selected search. The New Search Page would allow them to fill out all of the information required for a new search and then save their search. After creating a new search, results will be loaded, and the user will be taken to the View Search Results Page to view the results. Below is a picture of the

Select Search (Figure 10) and New Search Pages (Figure 11). Note that the Select Search picture (Figure 10) is shown using two example searches that have already been created.



Keyword(s)	View Search Results	Minimum Price	Maximum Price
sonic	View Search Results	25	35
mario	View Search Results	55	65

Figure 10

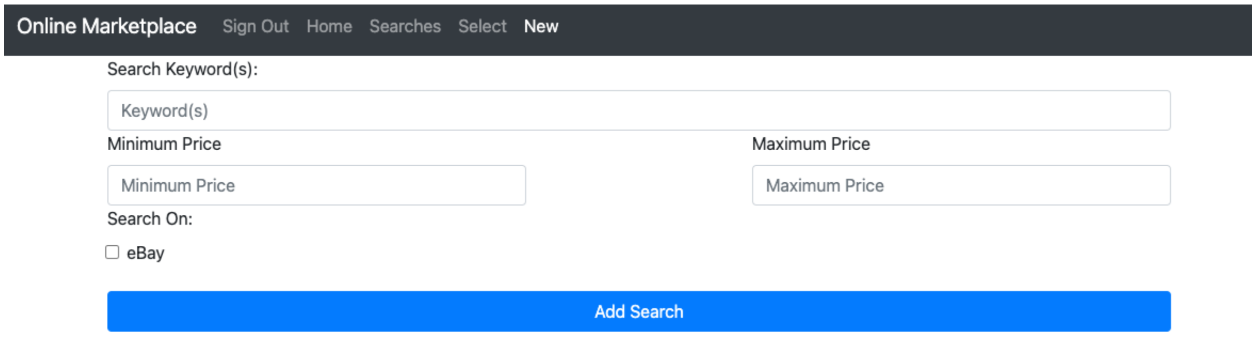


Figure 11

Both of the previous pages can be accessed from the navigation bar but can also be accessed through the Search Dashboard Page. Right now, the only thing the Search Dashboard does is give buttons to go to those pages. However, similar to the Home Page, this page is going to feature things like recent searches and recent hits. Below is the Search Dashboard Page (Figure 12 on the following page):

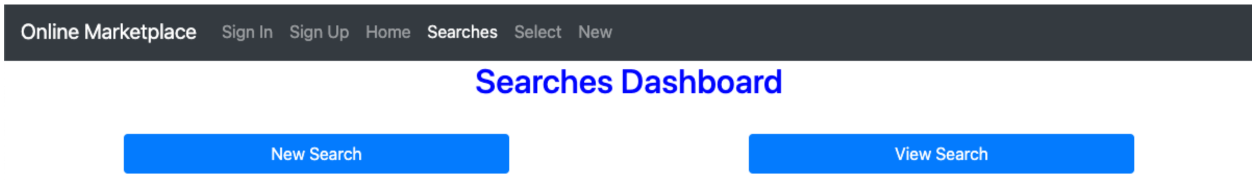
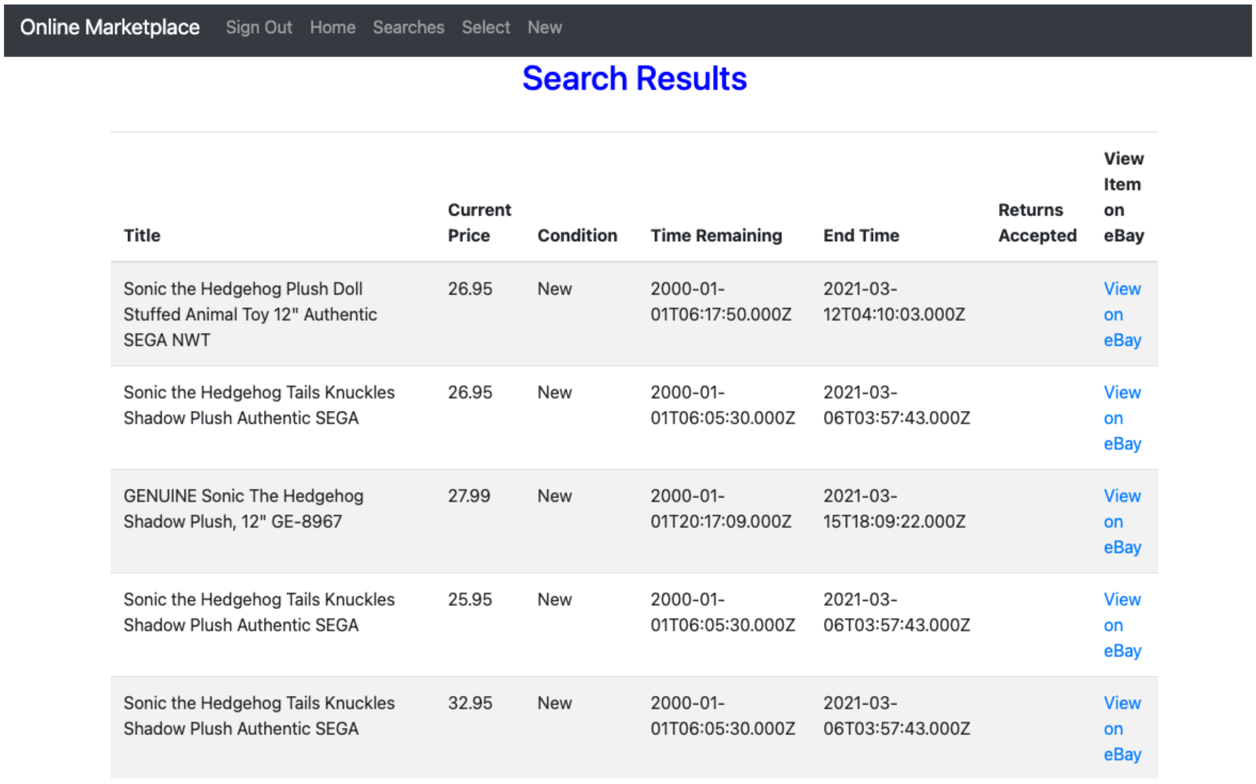


Figure 12

The last page is the View Search Results Page. As stated before, to get to this page, the user must create a new search or select an existing one. This will show them all of the relevant search results from that search, starting with the most recent results. On the following page is what the View Search Results Page looks like (Figure 13):



Search Results

Title	Current Price	Condition	Time Remaining	End Time	Returns Accepted	View Item on eBay
Sonic the Hedgehog Plush Doll Stuffed Animal Toy 12" Authentic SEGA NWT	26.95	New	2000-01-01T06:17:50.000Z	2021-03-12T04:10:03.000Z		View on eBay
Sonic the Hedgehog Tails Knuckles Shadow Plush Authentic SEGA	26.95	New	2000-01-01T06:05:30.000Z	2021-03-06T03:57:43.000Z		View on eBay
GENUINE Sonic The Hedgehog Shadow Plush, 12" GE-8967	27.99	New	2000-01-01T20:17:09.000Z	2021-03-15T18:09:22.000Z		View on eBay
Sonic the Hedgehog Tails Knuckles Shadow Plush Authentic SEGA	25.95	New	2000-01-01T06:05:30.000Z	2021-03-06T03:57:43.000Z		View on eBay
Sonic the Hedgehog Tails Knuckles Shadow Plush Authentic SEGA	32.95	New	2000-01-01T06:05:30.000Z	2021-03-06T03:57:43.000Z		View on eBay

Figure 13

Another critical aspect of the design was how I was going to structure the database. I knew I would need several different tables and that they were going to need to be in a hierarchical structure. On the following page is the table structure I planned to create (Figure 14):

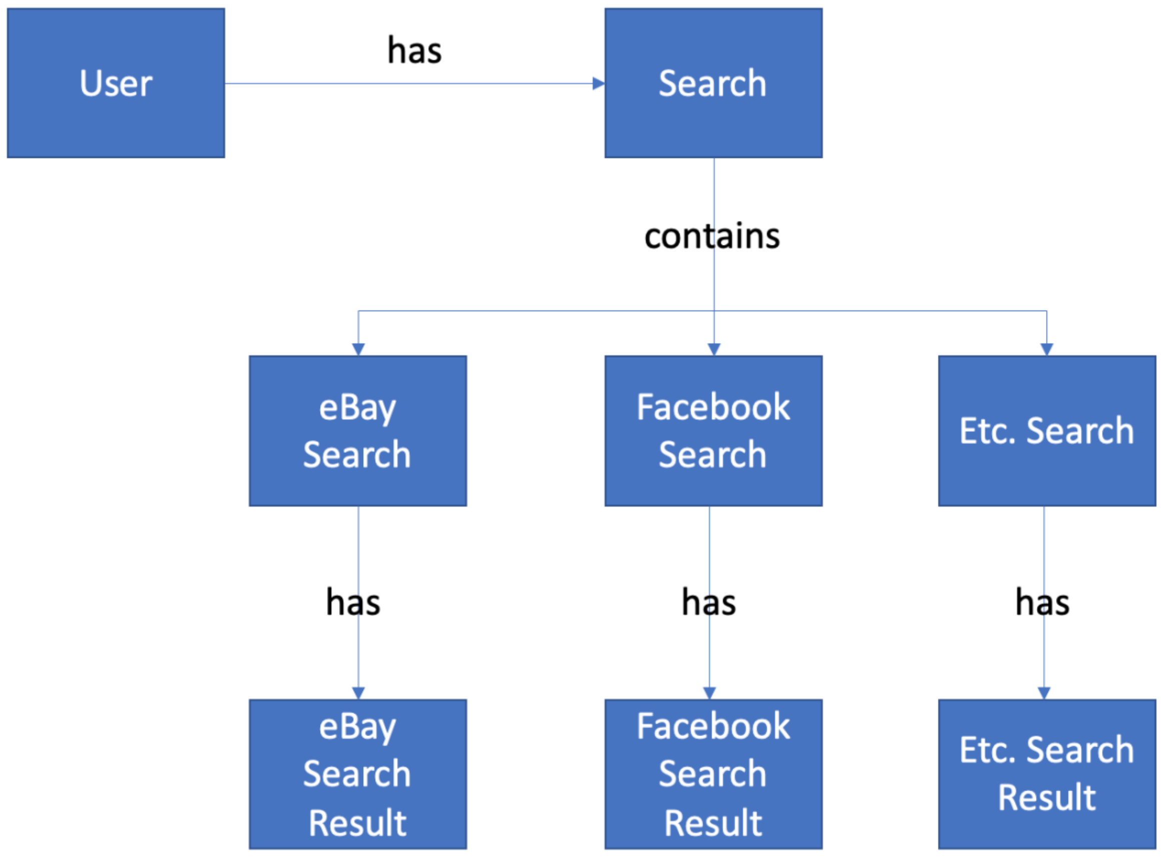


Figure 14

Using this structure, a user would be able to create as many searches as they wanted and could make them as specific as possible with each marketplace. While my research concluded that most search filters will be identical for each site, I wanted to make sure there was room for unique filters, such as eBay's "Buy It Now" filter, if a site provided them. Then, they would only have to remember one search and still get all of the possibilities of each site's filters. It would also be easier to get each site's search results if they were linked to a specific site. In addition to that, each search result can hold all of the specific information that’s contained in a search result. This structure would also allow for easy filtering of different sites and different searches for each user.

SET UP

The layout is rather simple, but the difficulty came in setting up and implementing everything. As stated previously, I decided to use the Ruby on Rails framework since I had used it during an internship. One of the benefits of using Ruby on Rails was its ability to use gems. Ruby gems are open-source libraries made by the community. There is currently a massive number of gems that can be very helpful with implementing difficult parts of the project. Gems are also usually extremely easy to install. I also chose to use AngularJS on the frontend. Not only had I used it at my internship as well, but I found that AngularJS provided a very nice compliment to Ruby on Rails with its MVC architecture. It also provided some JavaScript and HTML methods to reduce code duplication to almost zero.

I would also need a database to store the users, searches, and search results. Ruby on Rails provides a very easy option to select which database type to use. I chose to use the PostgreSQL database because of its community support. Postgres (a more common name for PostgreSQL) is still going extremely strong with updates from the community. It’s also scalable. One of the goals for this project was to be able to maintain this and make it last for a while. Postgres certainly provides a database that can meet those needs.

Lastly, I required something to help format my website. I chose to use Bootstrap. There were a few others I had checked but decided to go with Bootstrap because it offered the most options and I had used it previously for a few school projects before this one.

To get everything installed, excluding Ruby on Rails itself, I used a package manager called Bower. I did not want to include packages such as AngularJS and Bootstrap in a script tag due to the possibility of that causing lower performance. Instead, installing them directly into Rails was the optimal solution. Bower allowed me to install them directly into Rails with a few simple commands.

IMPLEMENTATION

BACKEND IMPLEMENTATION

I started implementing my idea by simply installing and launching the default Rails application. Once I had that up and running, I had to figure out how to store information securely. I didn’t plan on storing any sensitive information, but I wanted to make this project as realistic as possible. I wanted my user's data to be secure and started to research how to create secure database tables. After researching to figure out the best solution, I ended up using a gem called Devise to store user accounts. While I had started to create my own security implementations, there were a few reasons I stopped and chose to go with Devise. Devise is very convenient, handles everything for me, and is continuously updated with the best security practices. This would allow me to focus on the core of my project and still keep the data secure.

Then it came time to implement the database. Creating the tables was incredibly easy. Rails has a method to help with creating and modifying tables that makes doing so very simple. Rails also provides helper methods for connecting tables, which made creating the hierarchical structure a fairly straightforward process once I had researched the helper methods.

FRONTEND IMPLEMENTATION

Since AngularJS and Bootstrap were already installed, the most difficult aspect of the frontend was making sure I was keeping all of the files organized and everything consistent. I decided to make everything the user sees a component. Using components allows me to utilize AngularJS's MVC architecture efficiently. I was able to keep the view separate while all other parts of the component could be nicely combined into one file. Anything the user doesn't see (such as factories, services, and constants) could remain as one file.

BRINGING BOTH IMPLEMENTATIONS TOGETHER

While neither one of the frontend nor backend implementations were difficult to do individually, the difficulty came when trying to get them to communicate properly. Right off the bat, I was faced with the issue of using Devise. Devise uses ERB files to render its views, whereas I wanted to use AngularJS. I also did not want to convert Devise's views over to AngularJS in case Devise updates some functionality or interface concerning their ERB files. This is an issue because. Typically, Rails uses one application view that renders all of the pages. One application view can't render both ERB and AngularJS views. I ended up creating a second application view. That way, I had one to render ERB view files and the other to render AngularJS view files. Normally, Rails figures out what ERB view to show by looking at the URL. I took it a step back and made Rails pick which application view to render based on the view. Then, the selected application view renders the appropriate ERB or AngularJS view based on the URL. As a safety net, if the URL does not match any URL associated with my Rails application, it will render the Home Page. Below is an example of what the user will see when they load up the New Search Page and are not signed in (Figure 15):

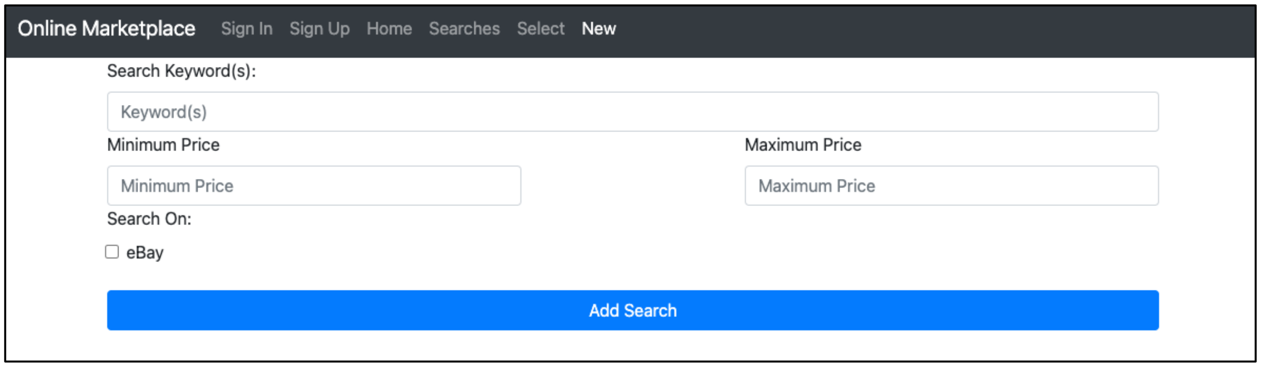
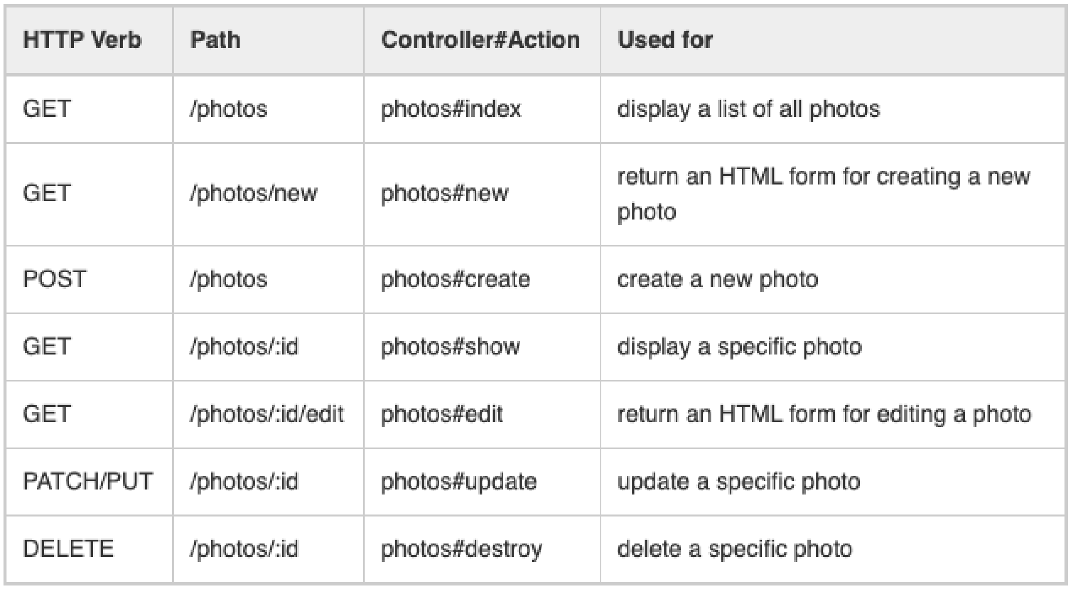


Figure 15

After understanding routing a little bit better, I moved on to creating states. Each state corresponds to a route. Meaning that when a non-Devise URL is given, the AngularJS template is rendered. Before rendering, the AngularJS uses the URL to determine which state to use. Once a state is determined, that state's component is rendered within the AngularJS view.

States are critical to using AngularJS efficiently. States can be assigned URLs, components and even resolve promises before loading a component. This is very useful when trying to organize the page layout of an AngularJS project as well as keeping everything as slim as possible. A default state can be selected too. That way, if the user accidentally goes to a page that does not exist, they are rerouted to the default state. Having a component for each state also means that the page itself can be created as a component, so there is less code duplication, as well as no need to specify a controller and a view (although that is still possible). Lastly, being able to resolve promises before the component loads is huge. Sometimes, a view will change depending on a variable (such as having the Sign In and Sign Up links instead of the Sign Out link if the user is signed out). Allowing for the component to know what that variable is before it loads to prevent a loading screen or a user interface switch a couple of seconds after the initial load completes is wonderful and helps to keep what the user sees simple and less confusing.

The last thing to bridge was a component making an API request to a Ruby controller. Routing, including internal API calls, is a huge part of the Rails framework. This is how Rails determines what view to load and what controller endpoints to call. By default, Rails provides a RESTful route for each resource. This provides seven default routes, shown below (Figure 16 on the following page):



HTTP Verb	Path	Controller#Action	Used for
GET	/photos	photos#index	display a list of all photos
GET	/photos/new	photos#new	return an HTML form for creating a new photo
POST	/photos	photos#create	create a new photo
GET	/photos/:id	photos#show	display a specific photo
GET	/photos/:id/edit	photos#edit	return an HTML form for editing a photo
PATCH/PUT	/photos/:id	photos#update	update a specific photo
DELETE	/photos/:id	photos#destroy	delete a specific photo

Figure 16

While the routing mentioned previously had been used to figure out what view to render, there are also internal API calls that can be made. These internal API routes use specific routes that are defined by the developer. This was a little different due to the HTTP calls having arguments and the responses containing JSON data. It was also difficult because AngularJS is not normally a part of the Rails framework. Therefore, AngularJS has no built-in way of making API calls. To amend this, I used AngularJS's $resource factory. The $resource factory allows AngularJS to easily issue HTTP requests to any URL with any parameters in a single line. It also returns a promise so that the code can continue executing while the HTTP request is being processed. This is an example of needing to create specific, developer-defined routes. For each API request AngularJS needed to make, a route had to be added.

Along with internal routing, a big part of this project involved connecting to external, public APIs. These calls were particularly tricky because they were all defined by other developers. They had so much documentation that it was often disorganized and hard to follow. It was also difficult because, while every aspect of the internal routes could be tweaked by me, the public API calls were very rigid. They had to be called a very specific way and returned very specific data. In the end, I ended up using two of Rails' classes to help with calls. Specific calls had to be made using Rails' Net HTTP class, and the data was parsed by Rails' Hash class.

DATA FLOW

To round out bringing the two implementations together, I needed to make the data flow from one to the other without any issues. With Rails, the data flow is pretty standard.

To explain, I'll use the example of a user entering a new search. The user first goes to the website. Upon arrival, Rails picks up their URL and checks it against known routes. Seeing that it doesn't match any defined routes, it reroutes the user to the Home Page. The user does not realize that they have been rerouted because this is all internal. The user already has an account but is not signed in, so they click on the Sign In tab on the navigation bar. This does not issue an API request of any sort but simply changes the URL to match that of the Sign In Page. When they arrive at the Sign In Page, they enter their email and password to sign in. This does execute an internal API request. It pings the User table in the database. It first checks to see if a user matches the email given. If it does, it attempts to match the password. If the passwords match, the user is signed in and is redirected to the Home Page. If the passwords don’t match (or no user with the given email can be found), the user is told that the given information does not match any known user, and to try again. If this happens, the user is not redirected and remains on the Sign In Page.

Now that the user is signed in, the user goes to the New Search Page. They do so by clicking on the New tab on the navigation bar. This, like when they clicked on the Sign In tab, simply changes their URL and does not issue any API requests. Now that they are on the New Search Page, they enter in all of the information. They enter a keyword, a minimum price, and a maximum price. They also select the eBay check box to search on eBay. After filling out all of this information correctly, they click on the Add Search button. Before issuing an API request, their information is checked to make sure it is valid. Things like missing the keyword section, having a minimum price that is greater than the maximum price, and having no sites to search on are all checked on the frontend first. If the frontend's checks are okay, an API request to the backend is issued containing all of the information the user just entered. The backend filters the parameters to make sure nothing malicious has slipped in and then reperforms the checks that the frontend did. If the backend’s checks fail, it will return the API request by simply notifying the frontend that the search could not be added. If the backend’s checks pass, it will create a new search.

When a new search is created, it is simply added to the database with the user's information. First, a parent search is created. In the database, this parent search is owned by the user. Since the user is searching on eBay, an eBay search is also created. The eBay search is owned by the parent search. Currently, the eBay search is the table that contains all of the user's search information. The parental search simply knows what external sites the search is searching on and matches the corresponding entries in each individual search table.

At this point, the external eBay API is pinged to gather the appropriate results. Once eBay's API responds, the results are collected and individually stored in the eBay search results table. Each of these search results is owned by the eBay search itself.

Once the database has finished saving all of the results, the backend lets the frontend know by returning the parent search's ID. The frontend receives this ID and automatically goes to the View Search Results Page and requests the search's results using the ID it was given.

Just like when the search was created, the frontend issues an API request to the backend using the parent search's ID as the only parameter. The backend receives this ID, makes sure a search exists with the given ID, and then proceeds to find all of the search results of the search. Once it has gathered all of the search's results, it returns them in the form of JSON data. The frontend receives the JSON data, parses it, and displays it for the user to see.

FUTURE WORK

There are several key issues that I believe need to be addressed when going forward. First and foremost is the lack of support for all but one marketplace. Other than eBay, there are no other sites that can be searched on through my site right now. The initial goal was to bring several site's items into the search results, but due to time constraints, that goal wasn't reached. Thankfully, however, this should be relatively easy, as all of the necessary tools and code are pretty much in place.

The website itself is also pretty bland and needs some improvement user interface. The Bootstrap that was used was minimal and added at the end, mainly so I could focus on making sure the core concept worked. The Home Page and the Search Dashboard Page are blank and repeat links easily given elsewhere, respectively.

Another issue, from a developer's perspective, is the lack of any testing. One of the quality assurance measures that was taken at my internship was that testing had to be done on every single line of code. This helped reduce bugs and errors before they went out to production for the everyday user to find. I would not be surprised if my code has some bugs in it that have not been exposed yet due to a lack of testing.

The code is also in desperate need of refactoring. Since I was learning most of what I did for this project for the first time, I have implemented some code in less-than-ideal ways. Naming conventions are not always used, there’s very little consistency between files and the file structure, and the backend has universal constants thrown in controller files.

Overall, I believe the project turned out well. I was able to make a website that encompassed my entire idea, albeit not to the fullest scope of the initial goals. I was able to learn so much about Rails, Ruby, and AngularJS. I felt that reading messy documentation and teaching myself about API calls allowed me to grow as a developer. I look forward to possibly presenting this project. To view the code, visit the Github repository at www.github.com/caxter99/honors_project.