Scholarship Technology Report
Prowler

NSN: 140404289

Link: https://prowler-nu.vercel.app/

Introduction

The rise in popularity of gaming has been nothing short of meteoric in recent years. What was once considered a niche hobby has grown into a global phenomenon and captured people around the world in all age groups. The gaming industry as a whole has also grown to become the most dominant entertainment industry, and was estimated to have generated over $180 billion USD in 2023. In fact, gaming is so dominant that its revenue exceeded that of film, television and music combined last year. One of the major drivers of this growth is the sale of video games across platforms such as Steam[1], Epic Games[2], and the Microsoft Store[3].

In February, I was unsure of what to do for my Scholarship Technology project and was brainstorming ideas. I was inspired after visiting PriceSpy[4], a website that compares product prices from various vendors, leading me to develop an idea for my project: a website that offers a similar service, but for video games. This idea formed the foundation for my project, which aims to provide users with the best deals on games across multiple platforms.

Initial Problem

The outstanding problem that I noticed when exploring the sale of video games was the lack of a centralised platform that allows gamers to quickly find and compare game prices across different vendors. If a customer wanted to purchase a game and ensure they were getting the best possible deal, they would have to visit multiple websites —such as Steam, Epic Games, and the Microsoft Store — and check the price on each one. This process is inefficient and time-consuming - especially so for gamers who frequently purchase games. Overall, the gaming industry lacked a comprehensive and user-friendly platform tailored towards helping gamers with this issue.

This gap in the market can often lead to missed opportunities and frustration, as consumers may purchase a game before realizing it was available for less elsewhere. Additionally, the inconsistency about how vendors go about sales can lead to users missing out on the best deals. For example, Steam will typically offer a large number of discounts during specific times of the year such as the Summer and Autumn Sales, whereas Epic Games tends to have more spontaneous discounts.

My Solution

My project aims to solve this outstanding problem by developing a website which provides the prices and details of video games across the two most popular vendors: Steam and Epic Games.

On my website, users would be able to efficiently find accurate prices and information about thousands of games on one centralized platform. Thus, my website would eliminate the need for users to manually visit multiple seller's websites. I would also automate the data collection process, meaning that information on my website would be always up to date and accurate.

My website would also provide a 'Browse' page where users could easily find new games to purchase. Alongside this, I would implement a robust and easy to use filtering system on said browse page, thus assisting users who had a rough idea of what they were looking for. Lastly, I would also track games on sale on both platforms and incorporate the percentage discount on the game's card if it was on sale.

---

[1] https://store.steampowered.com/
[2] https://store.epicgames.com/en-US/
[3] https://apps.microsoft.com/home?hl=en-US&gl=US
[4] https://pricespy.co.nz

Initial Planning

Since my project was quite robust and multifaceted, I decided to begin development by carefully mapping out each part of the process. I utilised a project planning table for this, with set dates and goals for each major aspect of my project. Looking back, this planner proved to be hugely beneficial, as it allowed me to have a clear goal in mind with a deadline. This led to me rarely getting sidetracked and working at a highly efficient rate.

Starting Date	Task
T1 Week 2	Begin initial planning phase
T1 Week 6	Finish the planning and begin work on the data collection method
T2 Week 1	Begin designing website in design tool
T2 Week 3	Receive ample feedback from stakeholders
T2 Week 4	Finish designing and begin development of website
T2 Week 7	Receive and implement first round of feedback for website (still in development) from stakeholders
T3 Week 4	Receive and implement second round of feedback for website (still in development) from stakeholders
T3 Week 6	Finish website and implement any final feedback as I need to leave enough time for my write up
T3 Week 9	Deploy final live version

I also broke my planning down into a couple of key stages in order to ensure I followed a structured and organised approach.

- I needed to identify what method to utilise for data collection on games.
- Select the most appropriate development tools for both the data collection and for the website itself.

Methods for Data Collection

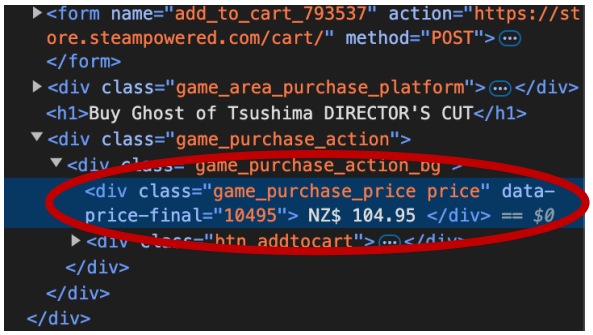
Collecting the data of video games is one of the most vital steps in my development process as providing this information is the primary purpose of my website. I needed a method which allowed me to collect mass data from both Steam and Epic Games, and ensured that this data was constantly accurate and up to date. For this goal, I did some research and narrowed my options down to three possible methods.

The first method was utilising public APIs provided by the vendors. This method initially stood out as a clear favourite, as many other websites I know of such as Steam Charts[5] utilise Steam's public API for similar purposes. Another advantage of using APIs is that they are seen as highly reliable since they are maintained by the platform itself, thus reducing the chances of misinformation or outdated data. These APIs are also selectively designed for developers, and are thus built to be easy to integrate into projects. For example, Steam's API has lots of clear documentation, sample queries and structured data formats (like JSON) which would surely help me in my development.

[5] https://steamcharts.com

However, using public APIs isn't all benefits and has some downsides also. Whilst Steam's API is widely used and comprehensive, Epic Games does not offer a widely accessible and easy to use API. In my research, I found Epic Games' API is designed for developers managing their own games on the website and general account information, rather than public game data. Their API is very limited and is not geared towards providing game details, prices or other data which I may need.

The second method was web scraping both Steam and Epic Games' respective websites. Web scraping involves extracting the HTML code from specific pages, then programmatically scanning this code and extracting the applicable information. A huge upside to scraping was the freedom and flexibility it allowed, as with this method you are able to extract any data you choose from the vendor's websites. This means I would be easily able to collect the more niche data I might use such as publisher, genres or game images without any issues. Since you are pulling data straight from the seller's websites, it also means that my website's data will be constantly up to date.



Although web scraping seemed like a great option, it also came with some negatives. Web scraping can, on occasion, come with legal problems if done irresponsibly (such as breaking terms of service). The laws around web scraping essentially state that it's completely legal to scrape all forms of information on websites, as it is public information. However, if you then resell or use that data to make a profit, you would almost certainly get in some legal trouble. Since my website will be completely non-profit, I won't have to worry about these limitations. Another potential issue which could arise is anti-scraping measures, where websites will randomise HTML class names, change them regularly, and utilise anti-bot checkers which scan the browser for bot-like behaviour which could all break the scraping program.

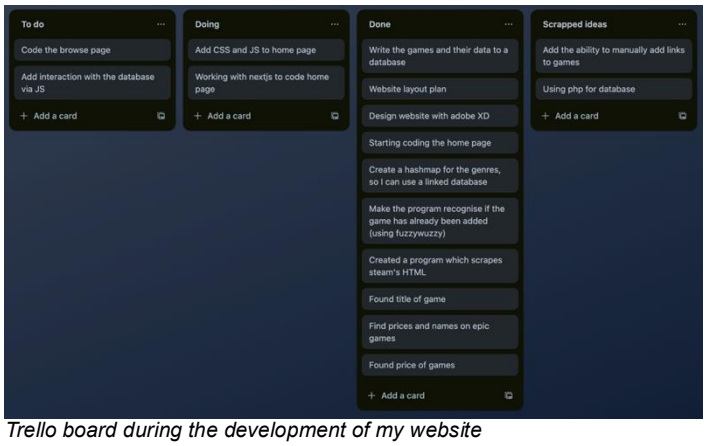
The third and final option was to reach out to both Steam and Epic Games and request access to their respective games databases. This is a fairly common practice within the industry as many companies will offer their data to third parties in exchange for promoting their platform, or some form of commission or payment. This method would undoubtedly provide the most accurate and up to date information since it would be the exact same information the sellers themselves use. It also comes with no risk of legal problems since the access to the database would be a mutually agreed upon partnership.

On the contrary, establishing such a partnership is incredibly difficult, especially for individual developers like myself. My website is completely non-profit, so I would be unable to do any sort of commission or payment deal. On top of this, due to the nature of my website comparing prices between vendors, the sellers may not be inclined to give me access to their database since it could potentially lead to less sales if their prices are not the lowest.

Overall, I came to the conclusion that the best suited method for my project was web scraping. Web scraping offers a lot of freedom and flexibility when collecting data, as well as being highly accurate and almost always updated. Whilst legal risks do pose a concern, I can mitigate all legal troubles simply by keeping my website non-commercial. On top of this, I can dodge scraping countermeasures by doing regular updates to my scraping programs when needed. In conclusion, web scraping stood out as a favourite between these 3 methods due to its accessibility, control, and reliability without the limitations which come with APIs or the need for any formal partnership.

Selecting Development Tools for Scraping

The first tools I considered when planning were project management tools. It is far too common for developers to run out of time and be unable to implement planned features within the deadline. Since I had already established strict deadlines for myself to complete each major part of my project, I believed it was essential to utilise project management tools to stay on track.
The first project management tool I knew I would be using was GitHub. GitHub is almost an essential tool for most developers and is used for version control and managing files, thus making it perfect for tracking my progress. With GitHub, I could ensure that my project remained organized at all times and I did not need to worry about ever losing my project since it was saved on GitHub's servers as well. It also meant I could transfer my progress and work to a different device since I could easily pull the main branch to any device and work on it there. It also provided a clear history of changes which I could scan over to quickly find bugs or errors in my code. In order to properly utilise GitHub, I ensured to commit and push at the end of every development session.



Trello board during the development of my website

Another development tool I knew wanted to utilise was Trello, specifically a board. With a Trello board I could keep track of the smaller tasks I needed to do throughout my project and have an overview of my progress. With Trello, you are able to break a complex project like my own down into simpler and easier to follow steps. Another reason I wanted to use Trello was due to my past experience in previous school projects with it, which meant I wouldn't have to waste precious development time learning how it worked and I also knew that it was helpful. On my board, I created three tabs initially: to do, doing, and done. I also ended up adding another 2 tabs later in development: potential ideas and scrapped ideas. I planned to initially add as many tasks as I could think of to the 'to do' tab, then move them around appropriately as I worked and planned to update the board at the end of every week.

For my web scraping backend, I also needed to decide what language I would be writing the scraping programs in. This decision was fairly important since different languages will offer different scraping libraries and some may be better suited towards my project. I was also more inclined to choose a language I had experience in (Python, PHP), so that I was not fighting an uphill battle trying to learn the language from scratch at the same time. After doing some research, I found that the most popular languages for web scraping programs were Python, Java and C#. From these three, it was typically left to personal preference for most people so I decided I would use Python as I was already very familiar with it. My familiarity with Python would end up being highly advantageous as it meant I could code the web scraping programs very efficiently.

Selecting Development Tools for Website

The development tool I decided to utilise for my website designing process was Adobe XD. Adobe XD is a program made specifically for designing user interfaces and wireframes. I had also used this tool before when designing websites, thus making it stand out as a clear choice. One of the main reasons I chose Adobe XD was due to its intuitive interface and wide range of features which allowed me to quickly test and refine my design ideas. Adobe XD also has a useful feature where you can add specific screen sizes that go from mobile phones to large computer displays, meaning I could also prototype the responsiveness for my pages.

Next, I needed to consider what tools I would use for my website development itself. I had already decided that I would be developing a website and not an app for this project, since the majority of end users will be gamers on their PC's or laptops. When considering what tools I might use to develop my website, I had to find the best middle ground in terms of complexity. If I chose an unfamiliar and highly complex tool, then I would likely spend a great deal of my development simply trying to learn said tool and losing out on actual development time. On the contrast, If I chose a very elementary tool, it's possible I might not have the necessary functionality I want when developing.

I knew that I wanted to utilise JavaScript on my website, because its versatility creates a better user experience due to its dynamic interactions and real-time updates. I also knew a small amount of JavaScript prior to my planning, making this choice fairly easy. I also had experience with PHP, however I found that JavaScript would be the superior choice between the two for my backend (as well as frontend). I found this as with specific frameworks, database connections can become rather seamless in JavaScript. Using JavaScript for my backend also meant my frontend and backend could be managed with a consistent and unified language, making it easier to maintain and expand in the future.

Choosing my JavaScript Framework

Next, I needed to identify what JavaScript framework I would be using. I knew from the start of my planning that I should use a framework because it would provide a more structured foundation. JavaScript frameworks offer pre-built tools and parts which simplify the more complex backend aspects of building a website, such as state management and routing. Thus, with a framework I would be able to better focus on the database connection and frontend aspects rather than losing my mind in boilerplate code. Additionally, many frameworks utilise modular coding, where codebases are separated and then pulled when needed (eg. navbar file which is then pulled into each page). Most modern frameworks also offer large communities around them and helpful documentation, which ensures that I would be able to find solutions for common issues easily.

When going about selecting a specific framework, I began by narrowing down my options. The framework I was to use had to be popular (so that there was sufficient documentation and community), free to use, and easy enough to learn. From these requirements, I was able to select three possible frameworks: React, Vue.js, and Angular.

React, whilst technically a library, is generally considered the most popular JavaScript framework. It was developed and maintained by Meta, continues to receive regular support and updates, and is available as a free and open-source framework. React was specifically designed for building user interfaces and leans into the modular coding ideology. Some of its main strengths lie in its ability to create reusable UI components, maintaining a website's state efficiently, and utilising a virtual DOM (document object model) in order to improve rendering.

Vue.js is another widely used framework which is quite similar in nature to React. It is also free and open-source, and is actively maintained. Vue.js is referred to as a 'progressive' framework as it can be integrated into projects already in development. It offers features such as a reactive data-binding system (updates UI as data changes automatically), component-based file structuring, and an API made to accommodate beginners. On top of this, Vue.js includes a wide range of additional tools and libraries (not included in base version) for developers like state management, routing, and SSR (server side rendering).

Lastly, Angular is a front-end framework developed and maintained by Google. Angular is a little more unique when compared to other frameworks, although still preferred and loved by a large community. It uses TypeScript by default and has a steeper learning curve due to its complexity. However, Angular websites get access to features which React and Vue.js

websites don’t, such as dependency injections, an object oriented approach, and two way data binding.

All three of these frameworks are solid choices for my project, however I concluded that I would use React for some defining reasons. Whilst Angular offers a robust and powerful framework, its nature of being entirely TypeScript based and its learning curve would require me to dedicate a significant amount of time to learning its ins and outs. Additionally, it’s a fairly complex framework which could lead to errors or complications which I wouldn’t get with the simpler frameworks. On the other hand, Vue.js seemed like an excellent choice for my project due to its beginner friendly API. Despite this, I ended up going for React as it was also beginner friendly and I felt that React’s flexibility, structure, and extremely large community was a slightly better fit for my project. On top of this, React also offers frameworks in and of itself such as Next.js, Gatsby, and Remix which provide even more helpful features and tools.

Another development tool I decided on utilising was Tailwind CSS. Tailwind CSS is a utility-first CSS framework that provides low-level classes directly in your HTML, allowing you to build custom designs quickly without writing traditional CSS. Firstly, Tailwind meant I could do all of my styling in the HTML part of the code, which would already be within a .tsx file due to React’s file structure, so I could essentially do all of the code for one page in a single file. This also meant that I could quickly update styling for specific parts of pages upon receiving feedback, which would happen frequently. Tailwind also makes doing media queries and general responsiveness much easier, due to its concise syntax for media queries and because it's on the same file so you can visualize how components will interact when shrinking. Finally, I made the decision to use Tailwind over Bootstrap because of its superior flexibility, as Tailwind allows for custom designs without the constraints of premade components (unique to Bootstrap).

The software I selected to develop my web scraping programs and website was Visual Studio Code (VSCode). I knew VSCode would make it easier for me to develop my outcome by providing an efficient and widely used platform which supports both frontend and backend development. One of VSCode’s most useful tools for me during development would undoubtedly be the built-in terminal, which would allow me to quickly run the python programs as well as run my website locally for testing without even needing to leave the editor. This would streamline my development process as it would allow me to test code very efficiently and update it when needed. On top of this, I also found the extensions available on VSCode were more than useful in past experiences. One such example is various TypeScript and JavaScript extensions which better power VSCode’s built-in JavaScript and TypeScript support such as JavaScript and TypeScript Nightly and ES7+ React snippets. Tailwind CSS IntelliSense would also be very useful as it provided me with auto filled suggestions when writing out Tailwind, which proved to be highly useful since this was my first experience with Tailwind, and it is a little bit different to regular CSS in syntax and general language. VSCode also has various quality of life functions like underlining errors and clearer indentation lines.

Development of Web Scraping Backend

Once I had completed my initial planning stage, I wasted no time and immediately began working on the web scraping programs.

Before I began actually coding, I needed to decide which web scraping library I would be utilising. I did some research and found that the most popular ones included BeautifulSoup, Selenium, and Playwright. After reevaluating my needs, I decided to go with BeautifulSoup due to its simplicity and beginner-friendly nature. My project required quick and efficient data extraction from large HTML chunks which BeautifulSoup thrives at. On top of this, I didn’t believe I would need any form of advanced browser simulation which is typically Selenium and Playwright’s strengths. I would eventually be proven wrong in this belief, however I will

elaborate on this later. Overall, I chose BeautifulSoup because its lightweight, has extensive documentation, and has a simple syntax.

First Iteration of Steam Scraper

I began by creating the python program to web scrape the steam web pages. My initial idea was to have an input for the steam game's page's URL and scrape that one specific page. My first version of the program would take the URL as an input, and then use BeautifulSoup to extract and parse the entirety of the page's HTML code. Next, I began playing around with BeautifulSoup to scrape certain parts of the page, such as the games' name and price. This can be achieved by using the soup.find function (built in BeautifulSoup function), and parsing a certain HTML id or class name to the function, as shown in the image. This meant that I needed to first manually inspect the HTML code for the specific web page, and then input the specific class or id of whatever I wanted to scrape. For example, if the game's name was stored in a class of id = "appHubAppName", calling soup.find(id="appHubAppName") would return the game's name, which I could then store in a variable (here its name_element).

```
+import requests
+from bs4 import BeautifulSoup
+
+URL = input("Enter URL: ")
+page = requests.get(URL)
+
+soup = BeautifulSoup(page.content, "html.parser")
+
+name_element = soup.find(id="appHubAppName")
+price_element = soup.find(id="game_area_purchase")
+price_element_1 = price_element.find(class_="game_area_purchase_game_wrappe
r")
+price_element_2 = price_element_1.find(class_="game_purchase_price price")
+
+print(name_element.text.strip())
```

However, I quickly encountered an issue when scraping a game's price. If the game I was scraping was on sale, the price's class name wouldn't be 'game_purchase_price price', it would be 'discount_final_price' instead, thus causing an error when BeautifulSoup was parsing for the incorrect class name. To fix this, I simply employed an if statement which would conditionally parse for class 'discount_final_price' if it existed, or parse for class 'game_purchase_price price' if it existed (there were no testcases where both of these classes existed, so this method worked properly for all games). I was also employing good programming practice throughout these early stages such as commenting and some basic error handling to ensure that my codebases remained readable and organised.

```
#checking if the game has a discount or not
if price_element_1.find(class_="discount_final_price"):
    discount_element = price_element_1.find(class_="discount_final_price")
    print(discount_element.text.strip())

elif price_element_1.find(class_="game_purchase_price price"):
    price_element_2 = price_element_1.find(class_="game_purchase_price price")
    print(price_element_2.text.strip())

else:
    print("failed")
```

Second Iteration of Steam Scraper

At this point, I could feed the program a Steam game's URL, and it would print said game's title and price, and was working with games on sale too. However, my end goal was a program which would automatically scrape this information (and more) for thousands of games, without any manual interaction or inputs. So, it was at this point where I began to consider how I could scrape multiple games with just one program.

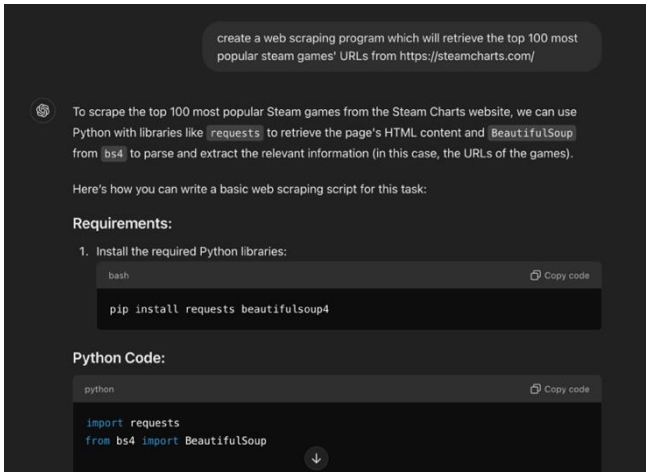
```
Enter URL: https://store.steampowered.com/app/264710/Subnautica/
Subnautica
NZ$ 44.99
kobe@Kobes-MacBook-Air Prowler % /usr/local/bin/python3 "/Users/kobe/Deskto
Enter URL: https://store.steampowered.com/app/1326470/Sons_Of_The_Forest/
Sons Of The Forest
NZ$ 43.59
kobe@Kobes-MacBook-Air Prowler %
```

Output of first iteration of Steam scraper at this point

My first idea of how I could achieve this was to store a large number of games' URLs in an array, and then iterate through this array and scrape each one individually. However, I still needed to find a method to generate this array, because if I manually filled out an array with the most popular games, I would have to constantly update it since new games would constantly be popping up. For this purpose, I decided I would use another web scraping

program, but for the purpose of gathering the Steam URLs for the most popular games.

I was going to use Steamchart's website which tracks number of steam users playing each game and orders them in popularity (also provides URLs to these games), and scrape it to generate the needed array. However, I thought that this process would likely be very similar to what I had already done in my scraping program, and made the choice to ask ChatGPT to generate the program. Part of my reasoning behind this comes from since ChatGPT will analyse other programmer's scraping algorithms, I may be able to learn new and better methods which I can then implement in my main scraper. Another part of my reasoning was because I thought the task was quite monotonous and not worth dedicating development time on.



ChatGPT's program ended up working as intended and successfully scraped the first 25 games on Steamcharts. The method it followed was similar to my own, however I did learn a couple of takeaways from the program, such as when BeautifulSoup finds a class with multiple subclasses, you can easily iterate through these classes with a for loop, which would prove to be highly useful in my final programs. Another thing I learnt was more effective error handling, such as checking that the initial request to access the steam page was successful (if so, response.status_code will be 200).

```
# Check if the request was successful (status code 200)
if response.status_code == 200:
    # Parse the HTML content using BeautifulSoup
    soup = BeautifulSoup(response.text, 'html.parser')

    # Find the table containing the top games
    table = soup.find('table', {'class': 'common-table'})

    # Extract game information from the table
    games = []
    rows = table.find_all('tr')[1:26]  # Exclude the header row
    for row in rows:
        columns = row.find_all('td')
        rank = columns[0].text.strip()
        name = columns[1].text.strip()
        link = columns[1].find('a')['href'].strip()
        current_players = columns[2].text.strip()
        peak_players = columns[3].text.strip()

        game_info = {
            'Rank': rank,
            'Name': name,
            'Link': link,
            'Current Players': current_players,
            'Peak Players': peak_players
        }
        games.append(game_info)

    return games
```

The only problem with the program was that it was only producing 25 games with their links (as shown in image below, links only include game's steam id since that's all you need for a steam game). This was because Steamcharts itself only displays 25 games per page. To fix this, I noticed that in the Steamcharts URL the program was scraping, there was a current page value. Thus, all I needed to modify in the program to scrape more than 25 games was to implement a for loop around the entire program, and then increase the value of the page number variable by one each time it reached the end of the for loop. With this modification, I could scrape as many Steam games' links as I needed.

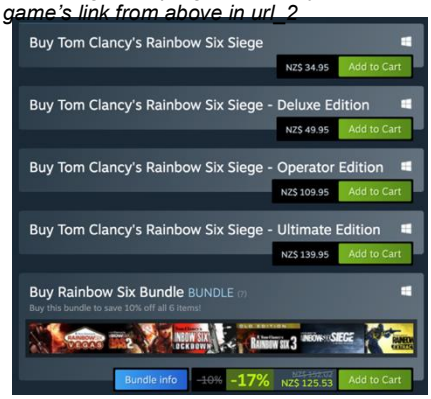
```
Rank: 1., Name: Counter-Strike 2, Link: /app/730
Rank: 2., Name: Banana, Link: /app/2923300
Rank: 3., Name: Dota 2, Link: /app/570
Rank: 4., Name: THRONE AND LIBERTY, Link: /app/2429640
Rank: 5., Name: PUBG: BATTLEGROUNDS, Link: /app/578080
Rank: 6., Name: Rust, Link: /app/252490
Rank: 7., Name: Baldur's Gate 3, Link: /app/1086940
Rank: 8., Name: Satisfactory, Link: /app/526870
Rank: 9., Name: Call of Duty®, Link: /app/1938090
Rank: 10., Name: Apex Legends, Link: /app/1172470
Rank: 11., Name: Warframe, Link: /app/230410
Rank: 12., Name: Grand Theft Auto V, Link: /app/271590
Rank: 13., Name: Wallpaper Engine, Link: /app/431960
Rank: 14., Name: Tom Clancy's Rainbow Six Siege, Link: /app/359550
Rank: 15., Name: Team Fortress 2, Link: /app/440
Rank: 16., Name: Black Myth: Wukong, Link: /app/2358720
Rank: 17., Name: Crab Game, Link: /app/1782210
Rank: 18., Name: Stardew Valley, Link: /app/413150
Rank: 19., Name: Once Human, Link: /app/2139460
Rank: 20., Name: War Thunder, Link: /app/236390
Rank: 21., Name: TCG Card Shop Simulator, Link: /app/3070070
Rank: 22., Name: NARAKA: BLADEPOINT, Link: /app/1203220
Rank: 23., Name: HELLDIVERS™ 2, Link: /app/553850
Rank: 24., Name: Dead by Daylight, Link: /app/381210
Rank: 25., Name: Warhammer 40,000: Space Marine 2, Link: /app/2183900
```

I then combined my initial scraping program with the ChatGPT generated program so that I could scrape multiple games in the same program. This worked perfectly since I could easily integrate them together – all I needed to do was add my initial program at the end of each for loop. This meant that once I had obtained the game's link, BeautifulSoup would then scrape that link and find the game's information which all worked as expected.

```
url_2 = f"https://store.steampowered.com{game['Link']}"
page = requests.get(url_2)

#parse the HTML content using BS
soup = BeautifulSoup(page.content, "html.parser")

#assigning elements of the page to variables
name_element = soup.find(id="appHubAppName")
price_element = soup.find(id="game_area_purchase")
price_element_1 = price_element.find(class_="game_purchase_action")
print(name_element.text.strip())
```

Screenshot of adding initial program. Clearly pulling the game's link from above in url_2

However, I quickly ran into a sizable issue with my entire methodology when testing it with a larger number of games. On each game's Steam page, there is often multiple price values; a price for the base game (what I want to scrape), and other purchasable offers like deluxe versions or DLCs (as shown in image). The problem arises in the fact that Steam will use the same HTML class names for all of the purchasable items on a page, thus making it impossible to identify what is the actual price for the base game. I spent a fair amount of time testing workarounds to this issue, but nothing was able to consistently identify the game's real price. Scraping all of the prices on the page and then selecting the lowest price didn't work, as some games offer DLCs which are cheaper than the game. Choosing the first one to appear in the HTML code also didn't work as in some cases, DLCs are listed above the base game. Additionally, this program very slow and inefficient due to visiting two pages per game scraped, increasing loading times and thus run time, which I needed to minimise since I would be scraping thousands of games. Unfortunately, I would need to completely rethink how I would scrape prices since visiting each games' page for the prices wasn't feasible.



Steam page for 'Tom Clancy's Rainbow Six Siege' with various offers

```
<h1>Buy Tom Clancy's Rainbow Six Siege</h1>
▼<div class="game_purchase_action">
  ▼<div class="game_purchase_action_bg">
      <div class="game_purchase_price price" data-
      price-final="3495"> NZ$ 34.95 </div> == $0
<h1>Buy Tom Clancy's Rainbow Six Siege - Deluxe
Edition</h1>
▼<div class="game_purchase_action">
  ▼<div class="game_purchase_action_bg">
      <div class="game_purchase_price price" data-
      price-final="4995"> NZ$ 49.95 </div> == $0
```

Same class name for different offers

Third Iteration of Steam Scraper

After looking around Steam's website for a bit, I found a page[6] (linked in footnotes) which listed games in order of popularity in an infinitely scrolling manner. On this page, each game's actual price is listed with it as well as a couple of other small details. After inspecting the HTML of this page, I found that I could use BS (BeautifulSoup) to select the container of the games, and then use BS's .find_all('a') (games are stored as 'a' links) function to return an array of all the games and their info. Then, I could iterate through this array and extract the necessary information including the link to individual game's page. Finally, I could create another instance of BS for the individual game's page, and then go about scraping the more specific data like description, developer etc, that can only be found on said page.

6 https://store.steampowered.com/search/?sort_by=_ASC&filter=topsellers&os=win&supportedlang=english

I needed to essentially start a new file to scrape this new page, as all of the class names would be different on this new page. Despite this, I was able to create a new functional scraping program very efficiently since the process was near identical to what I had already done, and the skeleton remained similar. In this new scraping program, I would first select the games container, and then use .find_all as mentioned previously to then iterate through each game's HTML section. Using the method outline above, I was able to retrieve each game's name, price, link, and description for now (output shown below).

```
#checking that the page successfully loaded
if response.status_code == 200:
    #parsing website's HTML
    soup = BeautifulSoup(response.content, 'html.parser')

    #creating a container for all the content
    container = soup.find(id="search_resultsRows")
    index = 1

    #for loop which loops through all the games since they are a links
    for game in container.find_all('a'):
        #finding name of game
        name = game.find('span', class_='title').get_text(strip=True)
        #container for price
        price_tag = game.find('div', class_='col search_price_discount_combined responsive_secondrow')
        #retrieving and error handling price
        price = error_handle(price_tag, "discount_final_price", "price")
        #getting link
        link = game.get('href')
```

Third iteration of Steam scraper main code section

```
kobe@Kobes-MacBook-Air Prowler % /usr/local/bin/python3 /Users/kobe/Desk
Game 1:
Name: Call of Duty®: Black Ops 6
Price: NZ$ 119.99
Link: https://store.steampowered.com/app/2933620/Call_of_Duty_Black_Ops_
Description: Call of Duty®: Black Ops 6 is signature Black Ops across a
 return of Round-Based Zombies.
```

Output of scraper at this point

Error Handling Function

At this point, I noticed that there were a large number of unusual listings on Steam which would cause errors in my program as they were missing a piece of information I was attempting to scrape. To fix these errors, I found myself manually coding in error handling for each piece of information I was scraping, however each time I did this, I noticed that the general structure of the error handling was identical. I decided I wanted to create an error handling function to increase my code's efficiency which is shown below. However, BS's find() function doesn't recognise the variables passed through the function as dynamic inputs. Since the variables are intended to be a placeholder for class names and containers, but BeautifulSoup requires a specific string or tag, the code fails to dynamically handle it. Thus, I would need to individually code error handling for each piece of data I extracted.

```
+#function to stop errors if a certain class doesn't exist
+def error_handle(container_1, class_name, variable):
+    if container_1.find('div', class_=class_name):
+        variable = container_1.find('div', class_=class_name).text.strip()
+    else:
+        variable = f"Failed to find {variable}"
+    return variable
```

The error handling function I mention

Addressing Infinite Scroll Issues

After this, I quickly ran into yet another significant issue. Despite not limiting how many games the program should scrape, it was only scraping exactly 50 games each time. After some experimenting, I realised that the cause of this problem was due to the infinitely scrolling nature of the Steam page itself. When BS retrieved the HTML code of the page, it would only receive the HTML for the first lot of games (aka before any scrolling down had been done). This was because the page only loads the additional HTML as the user scrolled down. Since BS was only capturing the static HTML at the point of request, it's unable access HTML after the initial load.

```
if response.status_code == 200:
    #time between each scroll
    scroll_pause_time = 0.4
    #find screen height
    screen_height = driver.execute_script("return window.screen.height;")
    i = 1
    #while loop to continue scrolling
    while True:
        #scroll down
        driver.execute_script(f"window.scrollTo(0, {screen_height * i});")
        i += 1
        time.sleep(scroll_pause_time)
        driver.execute_script("return document.body.scrollHeight;")
        #if condition for when website should finish scrolling
        if i > 5:
            break

    updated_html = driver.page_source
```

In order to resolve this issue, I would need to utilise another python library which can automate actions on the Steam page, also known as an advanced browser simulator. I did some research on this matter and found that a common choice for browser automation is Selenium. On top of this, I found a Stack Overflow article[7] from someone having a similar issue, and the replies contained information which taught me the basics of Selenium. At the start of my program, I initialised the WebDriver for Chrome (this is provided by Selenium), and then used a while loop to repeatedly scroll down the Steam page, thus loading more games. Lastly, I would request the page's HTML, which now contained the newly loaded games, and utilise BS from there. I also added a variable which controlled how many times the driver should scroll, so I could modify the number of scrolls during testing, whilst still gathering enough data when genuinely scraping games.

Finally, I began finalising my Steam scraper (for now) by adding some more code to extract the game's genres and developer from the individual game's page. This was seamless again due to my experience with BS at this point and because the structure for scraping is very similar for all data. I knew that I would likely want to scrape more data (things like publisher, images etc) about the games later, but I wasn't exactly sure about these smaller details yet, so I held them off for later. From here, I began development on the Epic Games scraper. The output of the Steam scraper at this point is shown to the right.

```
if soup2.find(id="genresAndManufacturer"):
    genre_container = soup2.find(id="genresAndManufacturer")
    genre = genre_container.find('span').text.strip()
    developer = genre_container.find('div', class_="dev_row", 'a').text.strip()
else:
    genre = "Failed to find genre"
```

Code to scrape genres and developer

```
Game 6:
Name: THRONE AND LIBERTY
Price: Free
Link: https://store.steampowered.com/app/2429640/THRONE_AND_LIBERTY/?
Description: Welcome to THRONE AND LIBERTY, a free-to-play, multi-pla
ansform into creatures to battle across land, sea, and air.
Genre: Action, Adventure, Massively Multiplayer, RPG, Free To Play
Developer: ACE Studios
```

Output of Steam scraper at this point

Epic Games Scraper

After looking around the Epic Games website, I was able to find a similar page[8] (linked in footnotes) to the one that I was scraping on Steam, which was handy as it meant I would be able to copy my initial skeleton to this scraper (I had two scraping program files, one for Steam and one for Epic Games). Unlike Steam which has over 100,000 games, Epic Games only has around 5,000 games on offer, which meant I would be able to scrape every single one rather than only the most popular. Next, I copied over my scraping skeleton to the Epic scraper, inspected the page's HTML and changed some class names accordingly in order to print each game's entire HTML code for testing.

Bypassing Cloudflare Restrictions

However, the first time I ran the program, the terminal immediately returned an error that BS couldn't find the specified class names. After printing the entirety of the HTML code for the page to diagnose the error, I realised that my program wasn't even making it to the Epic Games page, and was instead getting Cloudflare checked. This is a huge problem because Cloudflare was blocking my automated request and thus preventing the scraper from reaching its destination in the first place. On top of this, Cloudflare challenges are specifically designed to block automated traffic through IP rate limiting, captchas, and detecting browser behaviour, making it much harder for scrapers to collect data efficiently.

---

[7] https://stackoverflow.com/questions/20986631/how-can-i-scroll-a-web-page-using-selenium-webdriver-in-python

[8] https://store.epicgames.com/en-US/browse?sortBy=title&sortDir=ASC&category=Game&count=100&start=0

```
+import undetected_chromedriver as uc
+from webdriver_manager.chrome import ChromeDriverManager
+from selenium.webdriver.chrome.service import Service as ChromeService
+from selenium.webdriver.chrome.options import Options
```

Importing the undetected WebDriver

```
+options = uc.ChromeOptions()
+options.headless = False
+driver = uc.Chrome(options=options)
```

Selecting the undetected WebDriver

I worked to fix this problem by researching how other web scrapers have bypassed Cloudflare checks. I found the widely adopted method was by installing an addon for selenium called undetected_chromedriver. The standard WebDriver used by Selenium has 'botlike' parameters in its behaviour, which in turn makes it easy for Cloudflare to detect. On the other hand, the undetected driver patches these loopholes with more human-like fingerprints, and is able to bypass these checks every single time. The only downside to this method is that the pages on this driver load very slowly in comparison to regular BS requests. Overall, the benefits outweighed the downsides and I implemented this new driver and the scraper was working successfully.

Handling Pagination

```
+def scrape_epic_page():
+    start_value = 0
+    while start_value < 3800:
+        epic_url = f"https://store.epicgames.com/en-US/browse?sortBy=title&s
ortDir=ASC&category=Game&count=100&start={start_value}"
+        driver.get(epic_url)
+        time.sleep(2)
+        updated_html = driver.page_source
+        soup = BeautifulSoup(updated_html, 'html.parser')
+        container = soup.find('ul', class_="css-cnqlhg")
+        for game in container.find_all('li'):
+            if game.find('div', class_="css-rgqwpc"):
+                name = game.find('div', class_="css-rgqwpc").text.strip()
+            else:
+                name = game.find('div', class_="css-8uhtka").text.strip()
+            print(name)
+        start_value += 100
```

The next hurdle I had to overcome was Epic's pagination system. Each page displayed 100 games and my scraper needed to scrape every single game, meaning I would need to change the page number in the program. Since the page number was stored in the URL, I could deploy a similar method to what I did with my initial Steam charts scraper where I iterated through each page, increasing the page number by one each time, until I had reached the end. With this specific page however, instead of page numbers they used a 'start' value, so I needed to increase this by 100 instead with the start_value += 100 at the end of each loop iteration. With this implemented, I was able to retrieve every single game's name and price which Epic Games offered when I ran the program.

Finding a Practical Scraping Method

My next steps were to implement the scraping of individual games' pages for Epic Games so that I could extract more specific data such as the description, developer, and more. However, because I needed to use the undetected WebDriver since these pages were also secured by Cloudflare, it meant that I needed to manually load each game's page rather than simply retrieving said page's HTML source code (which is what BS does to save time). This resulted in extremely slow load times and scraping even 100 games took up to four minutes this way after some testing. Since I was to be scraping over 4,000 games, this means that each time the program ran, it would take almost 3 hours to finish. This simply wasn't feasible in the long term, prompting me to explore alternative approaches to bypassing Cloudflare, however all of them required the use of a WebDriver. Therefore, scraping each game's individual page as I did with Steam wasn't a practical option.

My workaround to this was simply to only scrape the data available on the page which contained 100 games. This data included each game's name, regular price, and, if applicable, sale price. I figured this data would suffice as I already could retrieve specifics from the game's Steam page. However, what this sacrifice did mean was that the scope of games I could scrape, and thus display on my final website, was slightly more limited. This is because I decided I would only add a game from Epic Games to my database if said game was also on Steam. I made this decision because for games exclusive to Epic Games, the only information I could gather was name and price which means the game's page would be almost empty. Thus my website would consist of games exclusive to Steam, and games on both Steam and Epic Games.

Addressing Title Differences

My next problem quickly followed, as one of my peers posed a question to me in class. He asked “What would happen if a game had slightly different names between Steam and Epic Games?”. I quickly looked around Steam and Epic Games and found an example where this was true; on Steam there is a game called ‘Tom Clancy's Rainbow Six® Siege’ however on Epic Games this same game was called ‘Rainbow Six Siege’. This possibility had minor but real implications as if such a case were to occur, then my initial plan of simply matching the names against one another to match the prices between platforms would be ineffective.

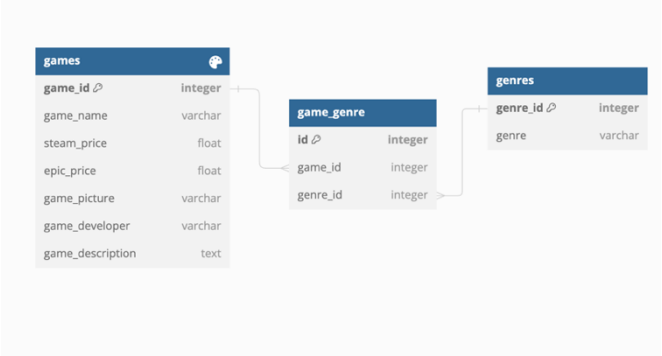
Same game, but with different names for Steam (top) vs Epic Games (bottom)

When considering how to potentially resolve this inconsistency, my first idea was to analyse the similarity of game’s name against any other names already in the database (I will expand on this logic and how I made the scrapers interact later). To achieve this, I imported a python library called ‘fuzzy wuzzy’ which analyses two strings and returns a decimal value for their ‘similarity’ (1.0 would be identical strings). Using this, I would be able to solve any inconsistencies where a game’s two names may differ by single characters or small amounts, however I quickly discovered a glaring issue with this system. Games with sequels would be considered the same. For example, the function shown below would consider GTA IV to be the same game as GTA V (thus falsely returning True) and this would clearly lead to errors and incorrect data. Overall, I decided to put a hold on this issue and deal with it after integrating a database connection.

```
+def is_similar(epicgame, steamgame, threshold=80):
+    similarity = fuzz.ratio(epicgame.lower(), steamgame.lower())
+    return similarity >= threshold
+
```

First Database Design and Connection

Before I integrated a database connection to both web scrapers, I drafted up a design for my database’s structure. Designing a database structure before creating it ensures that data inside of it will be logically organised, making it easier to manage and query effectively. It also allows for scalability, as a well optimised database which saves storage space can handle growing amounts of data more effectively. It was particularly important for me to plan my database’s structure as it would likely contain thousands of games, each with large amounts of data.



When planning my structure, I noticed that many games fell under multiple genres, which complicated my initial idea of using a linked list to save space for this field. Since a linked list could only store one genre per game, it wouldn’t suffice for cases where a game had more than one genre. I consulted my Technology teacher, who suggested using a linking table to address this issue. This table would connect the games and genres by storing pairs of game and genre IDs. For instance, if a game (id 5) had genres with ids 2 (Action) and 3 (Adventure), the linking table would store two entries for this game: (game_id: 5, genre_id: 2) and (game_id: 5, genre_id: 3), allowing each game to be associated with multiple genres. I used this idea and what I knew I was able to scrape to design the initial structure for my database shown above.

If I was going to follow this database structure, it also meant that I would need to have a table which contained all of the possible genres on Steam. Although only around 30 genres were actually widespread across thousands of games, there were over 400 genres total. At first I considered simply manually adding each of these genres, however this method would not only be tedious and painstaking, but it also meant that the genres could become outdated. To

create this table and keep it updated at all times, I decided I would utilise web scraping once again, as I was able to find a Steam page[9] (link in footnotes) which listed all of the genres which games could fall under.

I proceeded to create the database of structure outlined above on phpMyAdmin. I used phpMyAdmin for this simply because I only needed a local database for now, and we had used it in previous school projects, so I already had it set up and had some experience. On top of this, phpMyAdmin offers an easy to use interface when creating tables and databases, simplifying the process further. My next task was creating the genre web scraper, as outlined above. This process was very quick due to my experience with BS and because the page itself had a well organised HTML structure making it easy to scrape. I simply iterated through the genres container and stored each genre's name as a variable. In order to actually insert the genres into my database, I used mysql.connector into the program. I connected to my local database with the appropriate environment variables, and then wrote and executed the necessary SQL queries (with mysqsl.connector) to insert the scraped genres. I didn't need to worry about the genres' ids as I ensured that I used an auto-incrementing primary key for all of my tables.

```
import requests
from bs4 import BeautifulSoup
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from fuzzywuzzy import fuzz
import mysql.connector
import time


games_db = mysql.connector.connect(
  host = "localhost",
  user = "root",
  password = "2WS8YL8hqh988NxaPVP2iufPv",
  database = "prowler_games"
)
mycursor = games_db.cursor()

url1 = "https://store.steampowered.com/tag/browse/#global_492"
r = requests.get(url1)
soup2 = BeautifulSoup(r.content, 'html.parser')
genres_container = soup2.find(id="tag_browse_global")
index = 1
for genre1 in genres_container.find_all('div', class_="tag_browse_tag"):
    genre = genre1.text.strip()
    index += 1
    sql = "INSERT INTO genres (genre) VALUES (%s)"
    val = (genre,)
    mycursor.execute(sql, val)
    games_db.commit()
print(mycursor.rowcount, "details inserted")
games_db.close()
```

Next, I began working on implementing database connections for both the Steam and Epic scrapers. I started using mysql.connector again on the Steam scraper, and then writing an SQL query (shown to the right) which would be executed at the end of each iteration for each game. This query would simply insert variables I had preprepared earlier in the scraper into their applicable fields, and for now I simply inserted 0 or "N/A" for fields I hadn't prepared. In terms of the query, each %s will be replaced by the variable of value in the 'val' variable, by order. So the first %s will be replaced by the 'name' variable when the query is executed.

```
+
+           # SQL statement to insert the game's information into the prowle
r_games database
+           sql = "INSERT INTO games (game_name, steam_price, epic_price, ga
me_genre_id, game_picture, game_developer, game_description) VALUES (%s, %s,
%s, %s, %s, %s, %s)"
+           # values to insert
+           val = (name, price, 0, 0, "N/A", developer, description)
+           add_value = input("Add to Database? ")
+           if add_value == "y":
+               # adding values and saving them
+               mycursor.execute(sql, val)
+               games_db.commit()
+               print(mycursor.rowcount, "details inserted")
+
```

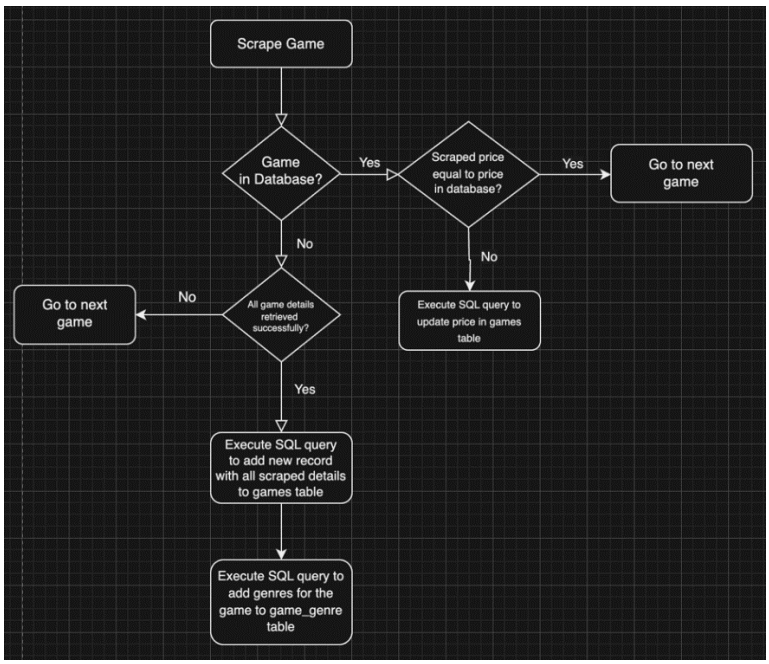
I also began to consider how I would manage the web scraping backend in the long term. I decided that I would initially add at least thousand games in an initial program; then, I would run the programs once per day, and it will check if the game is already in the database. If this condition was true, the program would only update the game's price (if it had changed), and if it was false then it would insert a new record with all the details (flowchart for visual explanation of this process below). To the right is a picture of my local database after inserting a few hundred records. I ran into a few problems whilst inserting the records, such as one record having a developer name which exceeded that field's character limit in the database. I fixed this by increasing the length in my database's settings. I also found that the description of the game which is varchar is limited to ~16000 characters, which might not be enough, so I changed the input type to text and set the limit at 30000. I choose text for this because varchar is more

Edit Copy Delete	1	HELLDIVERS™ 2	69.95	0	N/A	Arrowhead Game Studios	About This Game	The Galaxy's Last Line of Offence....
Edit Copy Delete	2	Dragon's Dogma 2	127.95	0	N/A	CAPCOM Co., Ltd.	About This Game	Note: There are other sets...
Edit Copy Delete	3	Counter-Strike 2	Free	0	N/A	Valve	About This Game	For over two decades, Coun...
Edit Copy Delete	4	Horizon Forbidden West™ Complete Edition	104.95	0	N/A	Guerrilla	About This Game	Join Aloy as she braves a majestic...
Edit Copy Delete	5	Tom Clancy's Rainbow Six® Siege	32.99	0	N/A	Ubisoft Montreal	About This Game	"One of the best first-per...
Edit Copy Delete	6	Lost Ark	Free	0	N/A	Smilegate RPG	About This Game	Embark on an odyssey for t...
Edit Copy Delete	7	Baldur's Gate 3	99.99	0	N/A	Larian Studios	About This Game	Gather your party and return to th...
Edit Copy Delete	8	Warframe	Free	0	N/A	Digital Extremes	About This Game	Confront warring factions ...
Edit Copy Delete	9	War Thunder	Free	0	N/A	Gaijin Entertainment	About This Game	War Thunder is the most comprehens

Screenshot of my phpMyAdmin local database populated with some games

[9] https://store.steampowered.com/tag/browse/#global_492

suited for short-medium lengths of text whereas text is specifically suited for very long lengths of text. Varchar is also stored within the table, making it faster for the shorter texts compared to text which is stored off table and is thus slower but more robust.



Flowchart for the steam scraper's logic

Checking for Games Already Scraped

The next priority was to accommodate my scrapers to update data if the game already existed on the database. Another benefit that came with utilising this method was that if a game was already in the database, I could skip the part of the program which scrapes the individual game's page. This is because all of the data that I extract from this page will, in an overwhelming majority of cases, never change. This means that when I am running the program once a day, its runtime will be hugely faster since the program has to visit less individual games' pages and can instead scrape data off the list of games page.

At the start of both scrapers, I added an SQL query which selected the every game's name and price (steam_price for Steam and epic_price for Epic) and stored these in an array. I then followed a similar process as what I did for the genres hashmap, except this time I was assigning each game's name to its respective price for the correct platform. This meant that I had a dictionary of all the games in my database corresponding with their prices on both scrapers.

For the Steam scraper, I simply used an if statement once the program had extracted all of the data. The if statement checked whether the game currently being scraped was already in the database (the str(round… around these values is to ensure that small differences eg 4.0 vs 4.00 are accounted for). If this was true, then it would again check, using the dictionary, if the price had changed, and if this was also true then the program would finally update the price. Otherwise, if the game wasn't already stored, then the program would insert a new record with all of the specific data. In this case, I also needed to

```
# checking if game already in database
if name in games_prices:
    # checking if price has changed
    if str(round(float(price), 2)).strip() != str(round(float(games_prices[name]), 2)).strip():
        print(f"{price} vs {games_prices[name]}")
        # sql to update price
        sql = "UPDATE games SET (steam_price, steam_on_sale, steam_normal_price) = (%s, %s, %s) WHERE name = (%s)"
        val = (price, on_sale, normal_price, name)
        mycursor.execute(sql, val)
        games_db.commit()
        print(mycursor.rowcount, "details updated")
```

```
else:
    # here, the game isn't in the database so we insert a new record for it
    # SQL statement to insert the game's information into the prowler_games database
    sql = "INSERT INTO games (game_name, steam_price, epic_price, game_picture, game_
    # values to insert
    val = (name, price, 0, "N/A", developer, description)
    if failed == False:
        # adding values and saving them
        mycursor.execute(sql, val)
        games_db.commit()
        print(mycursor.rowcount, "details inserted")
```

insert the appropriate records for the game's genres into the linked table.

Inserting Genre Data

I had to find how to identify each game's genres for the Steam scraper, what ids these genres corresponded to, and then insert these to the game_genre table. I did this by initially running a SQL query for the genres table which returned all genres and their ids. Subsequently, I created a hashmap (aka dictionary) which assigned each genre to its designated id (2: action, 3: adventure etc).

```
+genre_id = {}
+
+# get genre information to produce my dictionary
+sql = "SELECT genre_id, genre FROM genres"
+mycursor.execute(sql)
+results = mycursor.fetchall()
+for row in results:
+    # assigning the genre to its id in my dictionary
+    genre_id[row[1]] = row[0]
+
+print(genre_id)
```

```
genres = genre_container.find('span').text.strip()
genres_list = genres.split(', ')
```

```
genres_id_list = []
# iterate through the genres
for genre in genres_list:
    # add the id for the genre by searching for the genre in the dictionary
    genres_id_list.append(genre_id[genre])

# iterate through the values of genre id list
for a in range(len(genres_id_list)):
    # insert into the linkin table the values for the index of the game and the genre id
    sql = "INSERT INTO game_genre (game_id, genre_id) VALUES (%s, %s)"
    val = (game_id[0], genres_id_list[a])
    mycursor.execute(sql, val)
    games_db.commit()
```

At this point, I had a hashmap of all the possible genres assigned to their respective ids. From here, I used python's .split function and inputted (“, ”) as the parameter, since when I extracted the genres with the scraper, they were separated by commas (eg. Action, Adventure, Indie). This created an array of the genres named genres_list for each respective game. Next, I used a for loop to go through each genre in the genres_list array. For each genre, I searched the hashmap (called genre_id) to find its corresponding ID. I then added the ID to a new array called genres_id_list. Repeating this for all genres in the list resulted in an array of IDs. For example, if the genres were ["Action", "Adventure", "Indie"], the genres_id_list would be [2, 3, 1], where Action is ID 2, Adventure is ID 3, and Indie is ID 1. Here, I could simply iterate through the genres_id_list and insert each of its values to the game_genre database along with the game's id from the games table. I get this id value by doing a quick SQL query searching for the game's name. All code for this is shown above in the screenshots.

Updating Epic Prices

Lastly, I had to modify my Epic Games scraper to correctly update the prices, as it had only been scraping information and storing it as variables prior. I created a dictionary, which inserted games from the database and assigned each game's id to its name. Looking back, I would likely just use an array for this, since there was no need for each game's id. For each scraped game, I checked if it already existed in the dictionary, updating the epic_price column if it did; otherwise, I left the database unchanged. To handle differences in game names across platforms, I switched from the 'fuzzy wuzzy' library to list comprehension (the matches variable) to find partial matches. The matches variable loops through the steam_names dictionary, checking if the Epic Games name is a substring of any Steam game name. If a match is found, the corresponding index (game ID) is added to the matches list. Essentially, if a game's name on Epic Games was fully contained within a Steam name (e.g., Rainbow Six Siege is contained within Tom Clancy's Rainbow Six Siege), the program considered them the same game. Whilst this does mean that my program may miss some games if they differed slightly, it also means my website will be accurate for all game matches.

```
matches = [index for index, game_name in steam_names.items() if name in game_name]
if matches and not failed:
    # sql to update epic price
    sql = "UPDATE games SET epic_on_sale = %s, epic_price = %s, epic_normal_price = %s, epic_link = %s WHERE game_id = %s"
    val = (on_sale, price, normal_price, link, matches[0])
    mycursor.execute(sql, val)
    games_db.commit()
    print(mycursor.rowcount, "details updated")
```

Database interaction for updating Epic price on Epic scraper

Increasing the Scraper's Scope

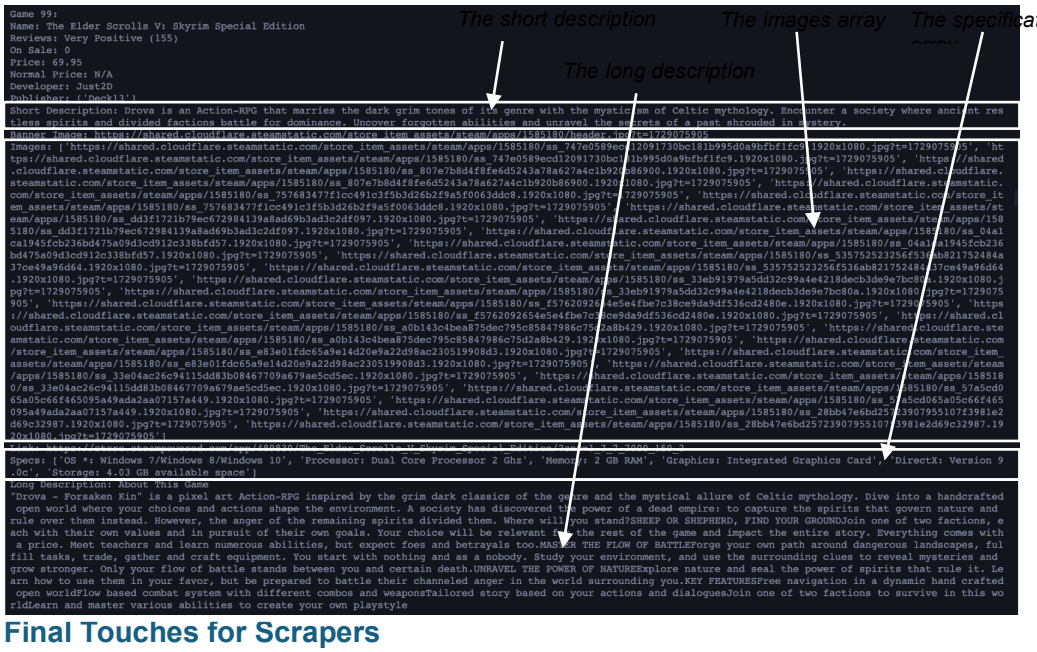
After this point, both of my web scraping programs were essentially complete, and only needed a few finishing touches. I showed some stakeholders my scraping programs and they recommended I should add more variables to scrape (ie. More things like images, developers etc) as this would make my site more attractive for users due to supplying more data. On Steam, I added scraping for reviews (overview for reviews), a 'on sale' field (0 for not, 1 for on sale), a normal price field (if a game was on sale, regular price stored here otherwise N/A), publisher, a banner, images, specs, and the page's link. These were all fairly simple to add to the scraper (code shown to right), due to my experience and well organised codebase. I also ensured to add error handling to each of these fields, as if the scraper couldn't find a specific variable, I would set a 'failed' variable to True, and then at the end of the program I ensured failed was false before inserting the data so that all of the records were valid. By doing this, it also meant that one game which caused errors wouldn't cause the entire program to die and stop scraping games, and I could easily go back and check what was causing the errors. This also helps me to recognise if the steam or epic games websites have updated their html code, and then I can fix the program appropriately.

```
# error handling and retrieving long description
if soup2.find(id="aboutThisGame"):
    description_container = soup2.find(id="aboutT
    long_description = description_container.find
else:
    failed = True
```

Majority of the new fields I added looked like the one above, where I could simply select the HTML class or id and then store it, however there were a couple trickier fields. The game's images were displayed in a carousel, so I needed to use an array to store all of the image's links in the database. The game's specification were also a cause for concern, since they weren't divided in the page's HTML. I simply decided I would store them in an array also based on each category, and then use JavaScript to properly format them on my website when I got to that.

A game's specifications section for Steam

After adding scraping for all of these variables, the end result of my steam scraper was being able to store extreme amounts of data about thousands of games. The information I am able retrieve for each game is shown and labelled (only the very long categories are labelled and put in boxes, the rest only take up one line and are titled on the left) below. This data is shown after being printed for each record in the terminal, and then added/updated accordingly in the database.



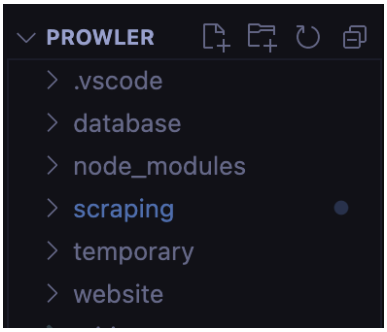
Final Touches for Scrapers

```
if price_container.find('div', class_="css-4jky3p"):
    normal_price = price_container.find('div', class_="css-4jky3p").text.strip()
    normal_price = normal_price.replace("NZ$", "").strip()
    on_sale = "1"
else:
    on_sale = "0"
    normal_price = "N/A"
```

Scraping code for the sale status and original price on Epic scraper

For the epic scraper, the only finishing touches I made was to also add the game's current sale status, original non-sale price, and the page's link. The reason I decided to store the game's original price if it was on sale was because I wanted to be able to provide sale percentages on my website. This was again quick to implement since I could just check if a certain HTML class (which was only there for games which had a sale) existed, and then if so set the sale status to one and find the original price from there. Around here, I also refined my error handling for the Epic scraper to be up to par with that of the Steam scraper. I also needed to actually update my database and create new fields to accommodate the new data I was scraping. This was nice and easy to do due to phpMyAdmin's interface, which made editing the database's structure possible with a basic knowledge of SQL.

From here, the final thing I added to my scrapers for now was more detailed comments which thoroughly explained each part of the programs and what they did. This meant that when I inevitably came back to fix mishaps, add more fields to scrape, or change the database, I would hopefully be able to navigate the code a little bit easier.



My overall project's file structure

Around this time I also separated my main project folder 'Prowler' (name of website) into a scraping folder and a website folder (currently empty), just so that I could better organise my files. I split my project folder into clear and separate folders for the website and the scraping as this will make my life easier when moving into the website development phase. I also had a database folder in the project folder which contained the SQL for my entire database, because it was currently only a local database on phpMyAdmin which meant it was best practice to store the SQL on my repository for backup purposes. I didn't need to worry about security of this since it was only the games database and didn't contain sensitive user information yet.

Conceptual Design Phase

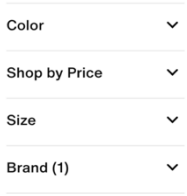
Upon initiating my website designing phase, I immediately began researching other websites which were similar in purpose to my own, as well as some random websites I have been impressed by. Looking around other websites before designing your own allows you to gather inspiration, identify effective design patterns, and avoid common pitfalls, ensuring that your site meets user expectations while standing out. I decided that my website would have a darker primary colour palette to create an immersive and visually striking experience, reflecting the mood and atmosphere often associated with gaming – thus I was predominantly browsing websites which utilised dark mode. Below is a summary of a couple of the main takeaways I learnt during this time.

From Nike's website, I learnt two key lessons: the importance of interactive elements on a page and allowing users to show/hide the filter menu. Nike's[10] website had many small but noticeable animations that added to the overall user experience, making the site feel dynamic and engaging. On top of this, their collapsible filter menu made for a cleaner interface, allowing users to focus on the content whilst still having easy access to sorting options. I noted these features and planned to implement them on my own site.
From shadcn/ui's[11] website (my eventual react component supplier), I gained an understanding of the vitality of minimalism in order to achieve a modern look. This site uses a

[10] https://www.nike.com/w/mens-jordan-shoes-37eefznik1zy7ok
[11] https://ui.shadcn.com/

simple colour scheme of a black primary, grey secondary, and white accent. I took note of where they used white and why, which was typically to put emphasis on the most important aspects of the page. I also thought highly of their navbar (shown below) due to its simplicity, ease of use, aesthetics, and subtle hover effects which addressed visibility of system status effectively.

I was also looking around a few popular website's filter sections during this time, as I would undoubtedly need to design one. My favourite sections I could find were from Nike and Stockx. I enjoyed using Nike's due to its user freedom, as I could extend and retract each category which made the menu overall easier to navigate. Stockx also had this feature, and another feature they utilised was scrollable categories. This means for categories which had a large number of fields, they made the options scrollable. This made the section better for aesthetics since one category wasn't taking up the whole section without sacrificing any functionality. Overall, by the end of my research, I possessed a large list of tips and notes for my own website's design which ended up being a huge help during designing and development.

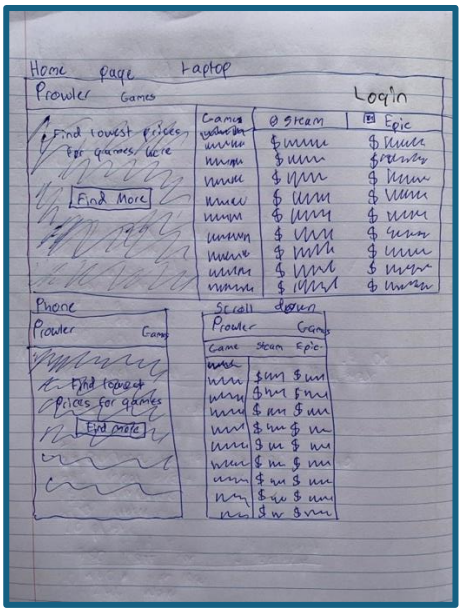


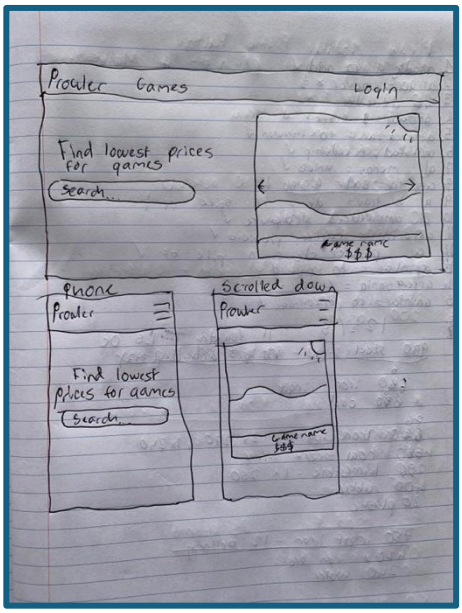
Nike's filter menu

I also knew I wanted to receive feedback from my peers throughout my entire design process to ensure that my design was user-friendly and appealing. Their input would help me identify areas for improvement and refine my work, allowing me to create a better final product.

Wireframes

The next step in my design process was to create some very elementary and simple wireframes to plan the layout of my website, specifically the home page. Wireframes are essentially blueprints of a website. I drafted up two different layouts for my home page, before showing my peers and asking for stakeholder feedback. I asked three classmates in total, and all three concurred that I was a talented artist and that the layout on the right was superior. Upon asking for a reason, they all said something along the lines of "the layout on the right is less convoluted", and "the image helps the layout on the right stand out more". I agreed with their reasoning and decided to choose the design on the right shown below.

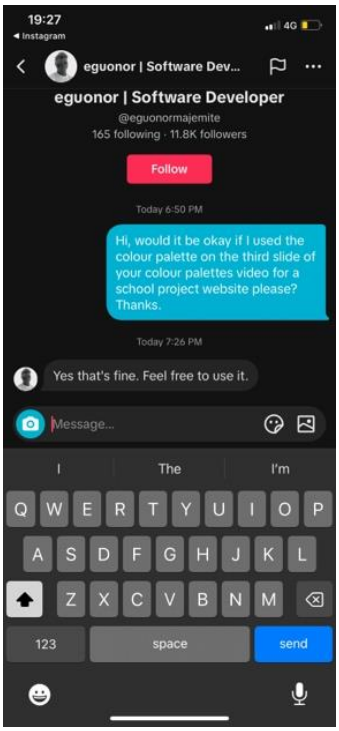


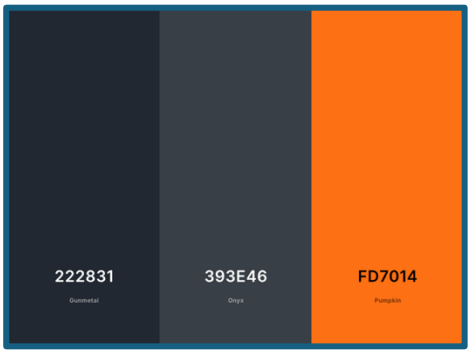


Adobe XD Interface Design

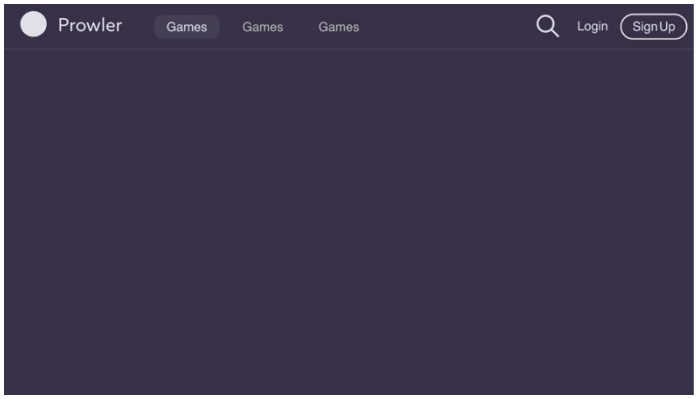
For the reasons outlined in my 'Selecting Development Tools' section above, I will utilising Adobe XD for prototyping website designs. The first decision I needed to make was to select fonts and colours, as both play a crucial role in establishing the site's overall feel and aesthetic. When choosing what fonts to use, I knew I wanted a clean and modern sans serif font to complement the overall sleek and minimalist design I was going for. With this kind of font, it ensured that all text would remain readable for every user adhering to accessibility. I also knew that I would limit the number of fonts on my site to a hard cap of 2; one for headings and one for regular text. With more than 2 fonts, a site seems to have too much going on and looks cluttered. With this in mind, I chose Azo Sans for my headings and Galvji for my body text (although I did change these later down the track).

For my colour palette, I knew I wanted to follow a similar style to shadcn/ui where I had a primary, secondary, and accent colour. Around this time, I actually stumbled across a social media video which contained a few colour schemes which stood out to me. I reached out to the creator (direct messages with them shown to the right) and asked permission to use the colour schemes to which they approved. The two schemes I noted down from this video were, in order of primary, secondary, accent: (#2A2438, #5C5470, #EFEFEF – left) and (#222831, #393E46, #FD7014 – right) which are shown below. Overall, I ended up choosing the left colour scheme as I felt the purple hues are more typically associated with gaming and they create a cohesive visual identity that enhances the overall user experience. Additionally, whilst I liked the idea of using orange as a unique accent colour, however it may have made the site have a less sleek feel due to its vibrance.





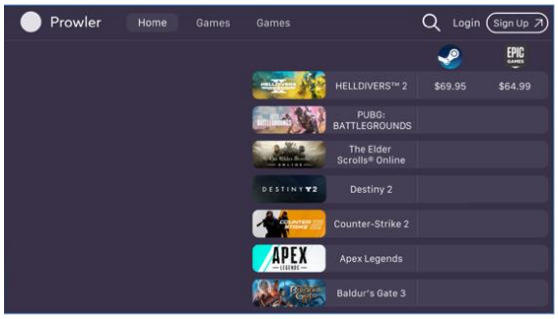
Navbar Design Process



At this point, I began actually designing in Adobe XD. I started with my navbar as it is one of the most vital aspects of a web page and is constantly used, so I needed to get it perfect. The initial design for my navbar was inspired by that of shadcn/ui. At the beginning of my design phase, I was still unsure about a few specifics of the page like exactly how many pages I would have or what my logo would be, so I simply used placeholders for these aspects. Shown to the right was my first design of the navbar. I used my accent colour for all of my text, which I got from shadcn/ui also, and used my secondary colour to indicate the user's current page which is

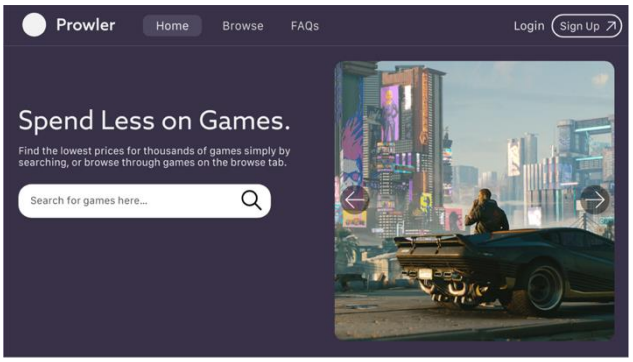
commonly used to indicate visibility of system status. I also used my accent colour for the outline around the sign up button to draw user's attention to it. The end goal of this effect was to get more users creating accounts as this would likely lead to superior user retention and more common visits. I also placed a search bar (to search for games) in the navbar for easy access as I understood that a large chunk of users would be visiting the site exclusively to search for one specific game. This placement made this task easier and more efficient to accomplish, and thus adhered to flexibility and efficiency of use also.

Home Page Design Process

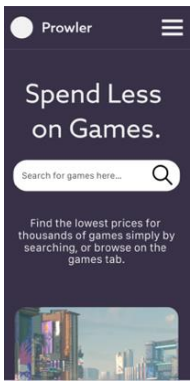


Next, I quickly drafted up the wireframe I didn't choose. I wanted to get a better idea of what it would look like to ensure I was making the correct choice. Simple wireframe drawings can be hard to visualise as an actual website, thus by actually designing this layout in Adobe XD I was able to make a better informed decision. I ended up with the design to the right, and asked my teacher for his thoughts, to which he reinforced that it was too cluttered for a modern and minimalist look on the home page. I also asked for some stakeholder feedback from my peers. One of them believed that the most popular games on display like this was a good idea because it could save users some time however the rest all agreed that the layout wasn't ideal for a landing page and thus I scrapped this layout.

By following this layout, I was able to create a very professional and modern look to my home page. To create this effective minimalist look, I kept text to a minimum and carefully established a visual hierarchy in the page using different fonts, font sizes, and colours. I ensured to make the search bar the centre of attention on the page, as for the reasons outlined above. Additionally, I also made the search bar in my accent colour of white, so that it stood out clearly to users as soon as they reached this landing page. However, because of this, I ended up removing the search bar from the navbar since having two search bars would cause confusion and serve as unnecessary clutter. I decided the carousel would display images from specific games, and then have a hover effect where that game's name and price would come up at the bottom of the image. The entire carousel would also serve as a link to that game's individual page. Although I was unable to achieve this in Adobe XD, my vision for the carousel was for it to automatically scroll through 5 – 10 games and loop back to the start, but it would also have manual controls as shown if it scrolled too fast for users.

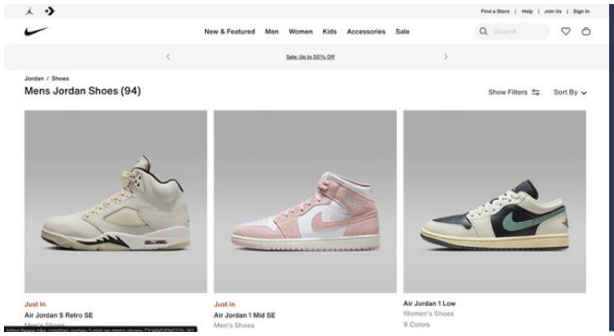


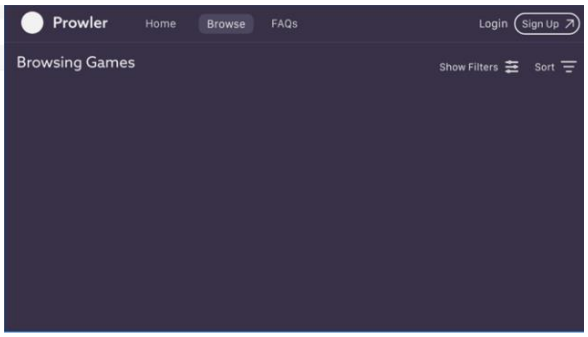
This design also follows consistency and standards to a high level, as I have placed the logo and name in the top left, account functionality in the top right, and then used left-aligned text on the left half of the page, which is proven to be easier to read. I proceeded by designing the same page for a mobile screen size. I decided I wanted to make designs for each of my pages on mobile too, as I valued usability for all devices. I was able to copy most of the regular design, however I found that my navbar wasn't large enough to accommodate all of the necessary links, so I integrated a hamburger menu in the top right, again following consistency and standards. This hamburger menu, upon being pressed, would cause a dropdown menu to appear which contained all of the stowed links.

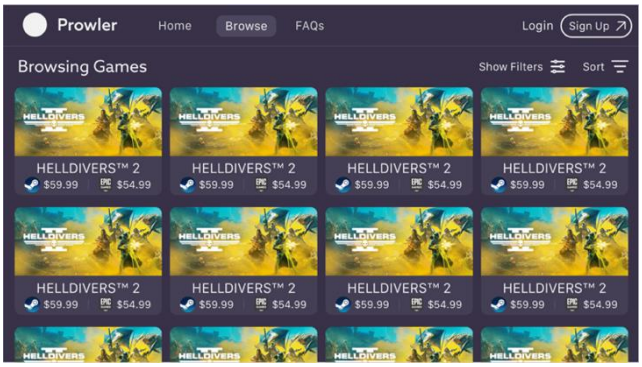


Browse Page Design Process

The next step was to begin the browse tab's design, for which I am taking inspiration from Nike's browsing page - I will be employing a similar layout for the filtering buttons. I believe having a specific design for mobile devices is very important as it is likely that many people will access the website through their phones, so I will also be designing a mobile view for this page. I also incorporated icons for the show filter and sorting buttons so I adhere to recognition rather than recall making the user-experience smoother.





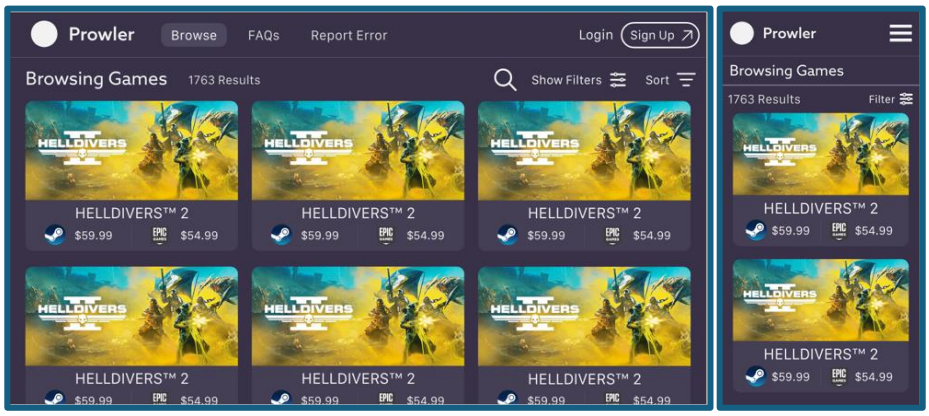


I also needed to design the body content for the games on this page. I knew I wanted a card design as based off of many other websites with scrollable products/games, a card system was most popular. With a card design, enough emphasis is placed on each individual game whilst still displaying a large number of games per screen length. On top of this, by using a card system instead of a listing design (as Steam uses), I can use large images of the game which helps users recognise games. Below was my initial design. When I was making this design, my first idea was to only display the name of the game below the image, then upon hovering the card, the text part would transition up to cover the card's image and show the price. However after tinkering with the design some more, I realised this would compromise the functionality of the page as users would want to be able to easily see the prices of many games as they scrolled (as this is the primary purpose of my site), so I drafted up this design shown which also displayed prices. In this design, I also used one game as a placeholder to save some time in this phase.

Here, I wasn't sure about the design as it didn't look quite right to me, so I again asked my peers for some feedback. I asked for some feedback from two of my friends and they both agreed that the page was too cluttered with four games in each row. They believed that having four games per row compromised the minimalist design and aesthetics of the page. I decided to make the page easier to navigate and use by changing the page to only 3 games per row, which would decrease the functionality of the site by a minor amount, but the improvement made by the extra white space and easier readability more than made up for this.

Additionally, one of my peers mentioned how they thought having a link in the navbar dedicated to the home page wasn't needed as I could simply use the 'Prowler' text and logo for this link. This change would reduce redundancy make better use of existing elements, and also align with consistency and standards as a wide majority of modern websites opt for this method. I also added a small 'X Results' text as it would likely be helpful for users to know exactly how many results are returned for their searches or filtering options, and this also boosts the visibility of system status for my site. Lastly, I integrated a magnifying glass icon which when pressed will slide out to a dynamic search bar (on type updates). This addressed both recognition rather than recall and match between system and real world

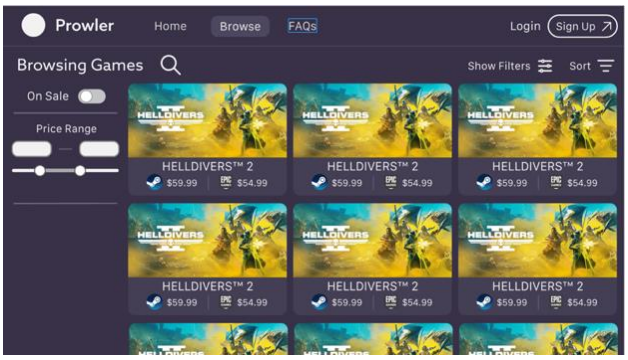
since people will typically associate the magnifying glass icon with searching, thus helping them quickly identify its purpose. I saw it as necessary to add this as even though users could search from the home page, many people may visit the browse page and then remember what game they wanted, or maybe they want to find a specific franchise of games. Overall, the end result of all of these changes is below.



When I was making these changes, I wanted to incorporate another page because I thought the navbar looked empty. However I found a meaningful page in a form of a contact page I temporarily named ‘Report Error’. On this page, I planned to have a form users could fill out to report any inaccuracies or misinformation on my website (emailed to me), which could become common due to my Steam web scraper’s logic. As I mentioned previously, the scraper will not visit individual pages for games already in the database to lower the program’s runtime. Thus, it will only have a chance to update prices and the game’s name, but none of the specifics like description, developer, genres etc which can only be found on individual games’ pages. After being contacted, I could go into the database and fix these mishaps to keep my data up to date.

Filter Section Design Process

Subsequently, I designed how the browse page’s filter bar would look, as I believed it to be one of core functionalities of my site. Taking inspiration from Nike’s filter bar, I incorporated a switch at the top to toggle games which were only on sale. Next, I began designing a price filter. I initially had the idea in my mind to utilise a slider for the price, since it felt more modern and intuitive than checkboxes, however I ended up going back and forth between these two options.

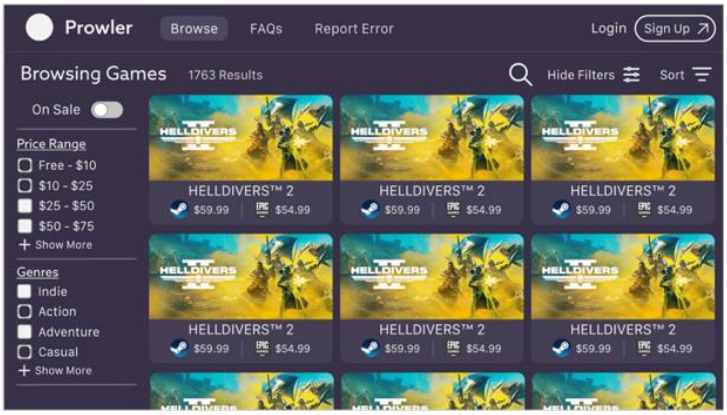


For price filtering, there is no widely agreed upon or clear best choice between a slider and checkboxes within the developer community – it’s generally seen as a case by case basis. To make this decision, I again revisited some popular websites, and noticed that Nike, Adidas, and Epic Games used checkboxes whereas Stockx, Airbnb, and Steam used sliders so this didn’t provide meaningful insight. I ended up doing some more in-depth research into the debate, and found an interesting comparison video[12]. The video demonstrates that whilst sliders may offer superior customisability, checkboxes get the (filtering) job done faster, which I believe to be more important for usability. Additionally, the users who want to filter by price on my site are far more likely to simply want to find a game within a price range (eg. cheap <$20, mid-price $20 - $40 etc) rather looking for games which are within a very specific

[12] https://www.youtube.com/watch?v=X_UprPZ1rCs&ab_channel=AndrewChart

range. These factors made me decide that I would use checkboxes for my design, but I figured I would probably come back to it when I'm programming the real thing.

Next, I added a genre filtering section, which I again used checkboxes for. This was relatively simple as they were formatted very similar to the prices, and using checkboxes for genres is essentially mandatory and used by every website. I did notice a small issue at this point however which was that I had over 400 genres in my database but actually including 400 genres in my filtering section didn't seem feasible unless I implemented a very long drop-down – and even that would need a search bar. To fix this, I decided I would only allow users to filter by the top ~10 most popular genres because anything outside of these was niche (genres like 'Detective' and 'Mahjong') and likely wouldn't be filtered by many users anyways. Despite this, I still had lots of genres to display, so in order to keep aesthetics without sacrificing functionality, I implemented a 'Show More' button at the end of each section. This meant that users could view some genres and price ranges by default and click 'Show More' to reveal additional options, keeping the interface clean, user friendly, and usable.



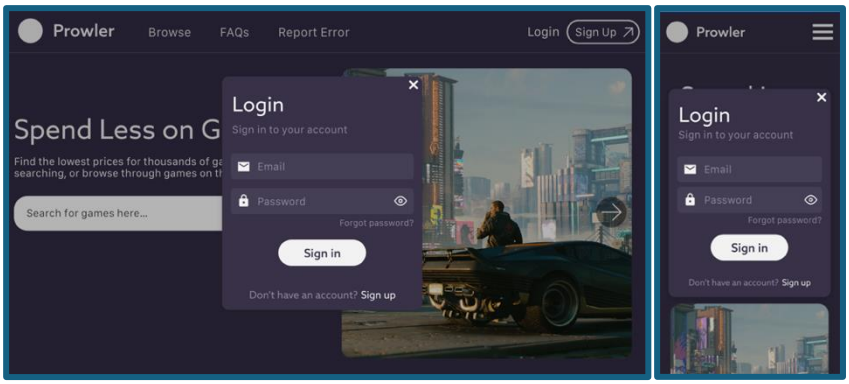
I didn't design a mobile view of the filter bar at the time, however I did have a vision for a popup menu which looked essentially the same as the filter bar above but took up most of the phone's screen. This design was based off what a large majority of other websites do for mobile filter menus as it allows users to still filter and sort on smaller devices without compromising the main content on the page.

Account Functionalities Design Process

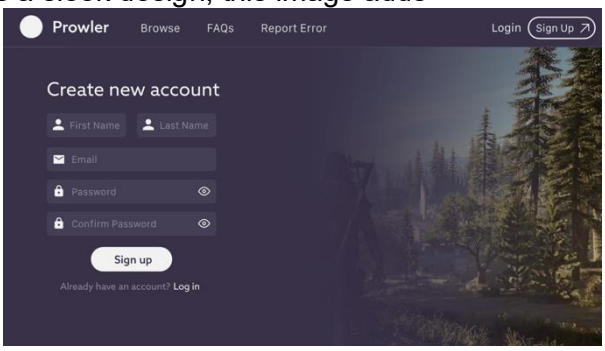
The next step in my interface designing process was to create the respective login popup and sign up page. I decided that I would only use a popup for logging in because it offers a fast and seamless process for users, which would be beneficial for them as they would likely be logging in several times (although I planned to use cookies, they could clear their cookies or login from a different device). On the other hand, the sign up process would require more details and is only a one time process, meaning a dedicated page made more sense for a smoother user experience. Thus, I began designing the login popup.

I started by again researching some other login pages or popups designs I could take inspiration from, specifically dark mode ones. I particularly liked two designs I found[13][14]. In my own design, I used a letter icon for the email and a lock and eye icon for the password. Through this, I am addressing recognition rather than recall and match between system and real world, helping users have a seamless experience when logging in. I also utilised colours to my advantage by marking the fields lighter, and making the login button stand out to guide users through the process. Lastly, I also darkened the background just to ensure users knew the popup was active and to emphasise it. It's also common practice to include a prompt to the sign up page on the login and vice versa in case of user errors.

[13] https://dribbble.com/shots/13928210-Dark-mode-Login
[14] https://nz.pinterest.com/pin/login-sign-up-dark-mode--678636237609973696/



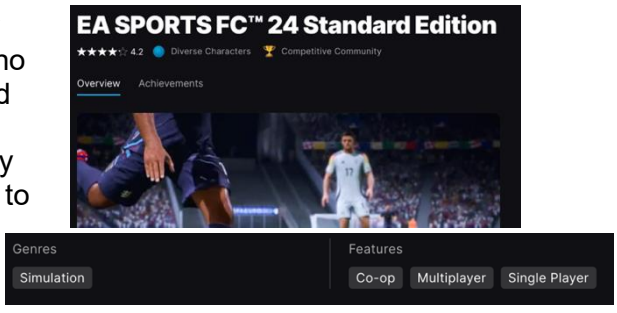
Once again, I utilised icons for each field in my sign up page. A person for the name fields, a letter for email, and locks with eyes for the password fields. I did this for the same reasoning as above since it adheres to recognition rather than recall and match between system and real world. This approach helps users by providing familiar visual cues that make the interface more intuitive, reducing cognitive load and making the process of filling out the form smoother and more user-friendly. I also added a faded game image to the right side of the page to fill in some blank space. While white space supports a sleek design, this image adds context and aligns with the site's theme without overwhelming the user, making the page more engaging. I didn't end up designing a mobile view for this page, however I did have a clear idea of what it would look like; I would simply remove the image as the window shrunk, and centre the sign up form. I asked my peers again for more feedback for both the login popup and sign up page. Despite asking four of my friends for feedback, none of them had any work ons for these pages. They all agreed that both of these pages were intuitive, easy to scan, aesthetic, and straightfoward.

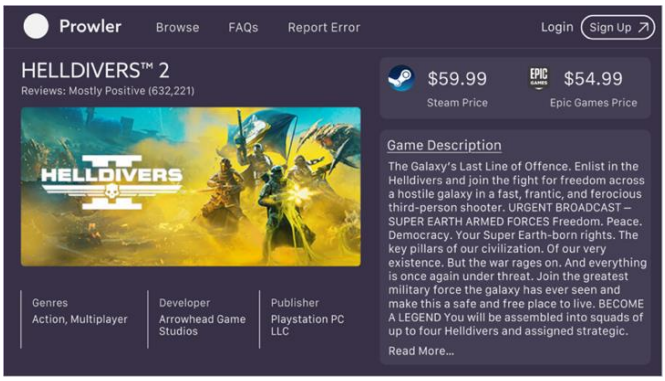


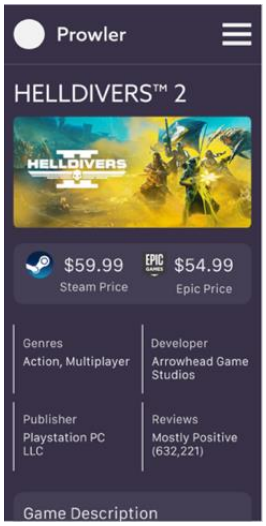
Game Page Design Process

The final page I wanted to design was the page for each individual game. I made the decision here that I would not design the FAQs or Report Error pages for a couple of reasons. Firstly, I was running low on time at this point. As per my schedule outlined in my introduction, I needed to finish this designing phase by the end of Term 2 Week 4 and by the time I had finished the login and sign up designs, it was the start of that week. Secondly, the FAQs and Report Error pages would be static pages which likely contained minimal content, and I deduced that it would be a waste of development time to design a prototype for them for this reason. Thus, I promptly began designing each game's individual page.

The individual game's page was a design which commonly had differences between different websites and there was no general layout for this, thus I would have to be original. I did end up doing some research and found I quite liked the design of Epic Games' individual games pages - specifically their title area and how they displayed genres (both shown to right). I thought they did an excellent job of aesthetics and minimalist design by keeping text to a minimum whilst still displaying adequete information.







When designing my own page, I knew I wanted a large image for the game, and I needed to emphasise the games' prices on both platforms since this was the main purpose of my site. My initial design took inspiration from the Epic Games page above in the genres, developer, and publisher areas. I thought this design followed aesthetics and minimalist design fairly well as there was lots of white space on the left and an established and purposeful visual hierarchy. However, upon asking a few of my peers for feedback, one of them mentioned that there was too much text on the page, particularly in the game description. While I noted their concerns, I didn't agree with the critique at the time and believed that the page was well-proportioned and not overcrowded however I did note the concern to potentially come back to, as I still thought it had some value.
Despite the issue above, the rest of the page is designed with usability in mind and creates a user friendly page. I also utilise Steam and Epic Games icons here for recognition rather than recall so that experienced users can immediately recognise which price belongs to which platofrm. Just below these however, I include some text saying 'Steam Price' or 'Epic Games Price' as help and documentation for less experienced users who may not recognise each icon. For this page, the mobile view was a little less intuitive as it wasn't immediately clear how I should organise the page into a single column design. I decided that I would place the title and image at the top, since these were obviously the most important aspects, followed by the genres, developer, and publisher section. I decided on this order as if the description was first, then it would be visible on the landing section of the page which as described above is bad for aesthetics. Mobile design is shown above next to the regular view.

Website Development Phase

My next stage of development was to code the website itself. For the backend of the website, I would be connecting to a database and thus as soon as the web scraping programs updated any information, the website would also thus keeping it up to date. As I explained in the introduction section, I would be developing my website utilising HTML, Tailwind CSS, JavaScript, TypeScript, and React as a JavaScript library. Since I would be using React, it would also be wise to utilise a React framework, such as Next.js, Remix or Gatsby. These frameworks all come with pre-built functionality, optimized performance, and will typically allow you to spend less time connecting tools by writing boilerplate code.

Selecting my React Framework

Before deciding which React framework I would use, I decided to do some research on each of them to ensure I choose the best tool for my project. Firstly, Remix offers structured data loading, which allows developers to fetch data for multiple routes in parallel. This essentially improves streamlines the data fetching process and improves performance. It also supports nested routes, as it simplifies nested views and data dependencies which can be helpful for complex projects. Lastly, Remix offers built-in form handling features, meaning you can easily implement form submissions and validations without external libraries which could prove useful for my own site due to the Report Error page's functionality.

Gatsby is extremely flexible and allows you to create almost any web application. This is due to its extensive plugin ecosystem and third-party services. It also utilises powerful image optimization features which automatically process images for faster loading times and superior performance. However, Gatsby truly excels in static site generation, which since my

site will be pulling from a database, doesn't make it ideal for me. It's also seen a massive decline in popularity as of recent years making it have a smaller community and less frequent updates.

On the other hand, Next.js has become extremely popular and is seen by many as the primary choice for beginners. It offers robust server-side rendering, which greatly improves search engine optimisation and much faster initial page loads than a framework like Gatsby. Next.js also allows for the creation of API routes within the project, which makes server-side logic and database connections much simpler. This also means you don't need to worry about the hassle that is external servers, unlike Remix. Additionally, Next.js is developed by Vercel, who provide a powerful cloud platform made for frontend developers. By choosing Next.js, I could also have access to a tool which allows developers to deploy live versions and have a cloud database in the same ecosystem as my framework, which simply would streamline many aspects of development. These services would be especially helpful for my project as I valued user feedback so a live testing server would be essential. Therefore, the choice was clear to me that Next.js was the best framework for my project.

The first step of my development was to correctly setup a project file, thus I created a 'website' folder within the 'Prowler' folder. Next, I followed Next.js' documentation[15] for how to get started. This began by navigating to my website folder in my terminal, before initializing a new Next.js project by running 'npx create-next-app@latest' to scaffold the project structure, and ensured to follow the prompts for naming and configurations. After that, I installed the necessary dependencies like React and ReactDOM by running npm commands in my terminal such as 'npm install react react-dom' and so on. I also set up Tailwind CSS in the folder by installing it via npm, configuring it with Tailwind's documentation, and creating a tailwind.config.js file. This file contains customisable classes for Tailwind, for example if you add 'big': 40px; to this file, and then set a margin to big, then it will be 40px which can be very useful if you are using the same colours or fonts many times as I will. I also needed to add a few Tailwind directives in my global CSS file. Finally, I confirmed that everything was correctly set up in the website folder by running the development server with the command 'npm run dev'. The server started successfully, and upon visiting localhost:3000, I was greeted by the example page, indicating that the setup was complete and that I was ready to begin development.

Navbar Development

I began development of my website by working on the navbar. For this, I made a nav.tsx file in my components folder so that I could easily import it to all of the subsequent pages I made without needing to have multiple navbar files - this is another benefit of using React. This is also multipurpose as if I want to change the navbar, I will only need to edit it in one file and it will change across all pages. I designed a prototype logo in photoshop which was basically just a copy of React's logo (however I would go on to make my own) and imported it into the public folder. I then used the 'justify-between' CSS property to place the links to the left and the account functionality to the right. I then spent some time editing margins and sizing to ensure I was satisfied with the layout. This was a very efficient process due to React's auto-updating feature where if you have a development server as I did, it would automatically update when a file is saved, meaning I didn't even have to reload the page to test CSS properties. I did end up changing some colours on the navbar from my initial design however the general layout remained identical. My first finished prototype for the navbar is shown below, however the navbar would change several times throughout the project as I received feedback. During this time I was also playing around with a few primary colours because I found that I didn't like the initial primary hex I was using in my designs. I wanted to go for a darker primary hex as people will typically associate gaming with very dark blue/purple hexes or black.

[15] https://nextjs.org/docs/getting-started/installation

Home Page Development

Next, I began developing the home page which was also the landing page for my website. It was crucial to create a design that was not only visually appealing but also user-friendly and functional, as it would be the first impression for visitors. As I explained in the conceptual design phase, I was going to implement a carousel to display some featured games on this page. In order to cut down on development time, I opted to utilise a pre-built carousel component from a React component library. When choosing what component library to use, it mostly comes down to personal preference for the library's style and overall look, since most of them use very similar building blocks and function identically. For example, almost all of the carousel components I could find were built using Embla's carousel library[16], and only differed in styling. After looking around a few component websites, I chose to use shadcn/ui for a few reasons. These reasons included that the style of their components matched what I was going for in my own site, it was free to use, and their components were highly customizable with extensive documentation and guides. I would continue to use shadcn/ui for all of my components so that my site had a consistent feel to it.

Image Carousel Component

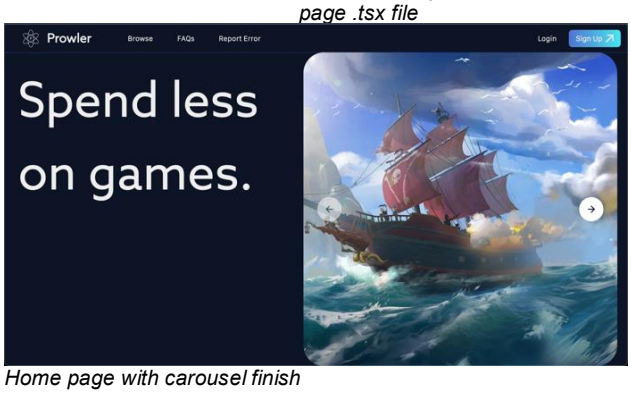
Importing shadcn/ui's carousel component was a fairly simple task due to this documentation; I just had to run 'npx shadcn@latest init' to initialise a ui folder in my components folder (file structure was /components/ui/carousel.tsx) and then run 'npx shadcn@latest add carousel' to actually install the carousel's file in the ui folder. Next, I could simply import the file from the components file into my home page and use it there. I could then add a <CarouselItem> tag for each image in the carousel and then import the images I was using for the games and assign them accordingly. However, upon doing this I immediately noticed I could make my code more concise and efficient by establishing an array at the beginning of the codebase which contained the images I was going to use. Then, instead of having five (I had five game images at this point) <CarouselItem> tags, I could use {carouselImages.map((image, index)… (this function iterates through each item in the carouselImages array, similar to for loop) around this tag and then dynamically code the carousel's items in by using the image and index variables (eg. image src= {`/images/${image}`}). This meant I only needed one <CarouselItem> tag, making the code cleaner, more efficient, and easier to update by simply managing the images through the array. Lastly, I customised the carousel component by using an autoplay plugin (automatically scrolls through images), and making this autoplay stop when a user was hovering over the carousel.

```
+const carouselImages = [
+  "seaofthieves.jpg",
+  "cyberpunk.jpg",
+  "tsushima.jpg",
+  "reddead.png",
+  "tsushima.jpg",
+]
```

Array being mapped

```
<Carousel
  plugins={[plugin.current]}
  className="self-center size-carouselsize"
  onMouseEnter={plugin.current.stop}
  onMouseLeave={plugin.current.reset}
>
  <CarouselContent>
    {carouselImages.map((image, index) => (
      <CarouselItem key={index}>
        <Card className="bg-transparent">
          <CardContent className="flex p-0">
            <Image
              src={`/images/${image}`}
              alt="Game Image"
              className="rounded-4xl"
              width={1300}
              height={1300}
            />
          </CardContent>
        </Card>
```

Carousel component in the home page .tsx file



Home page with carousel finish

More Design Decisions

Next, I decided that I didn't like the font I was using (Azo Sans), so I did a little bit of research on best simple and modern fonts – as to follow aesthetics and minimalist design. I also knew I should select a simple font because majority of professional websites use minimalist fonts

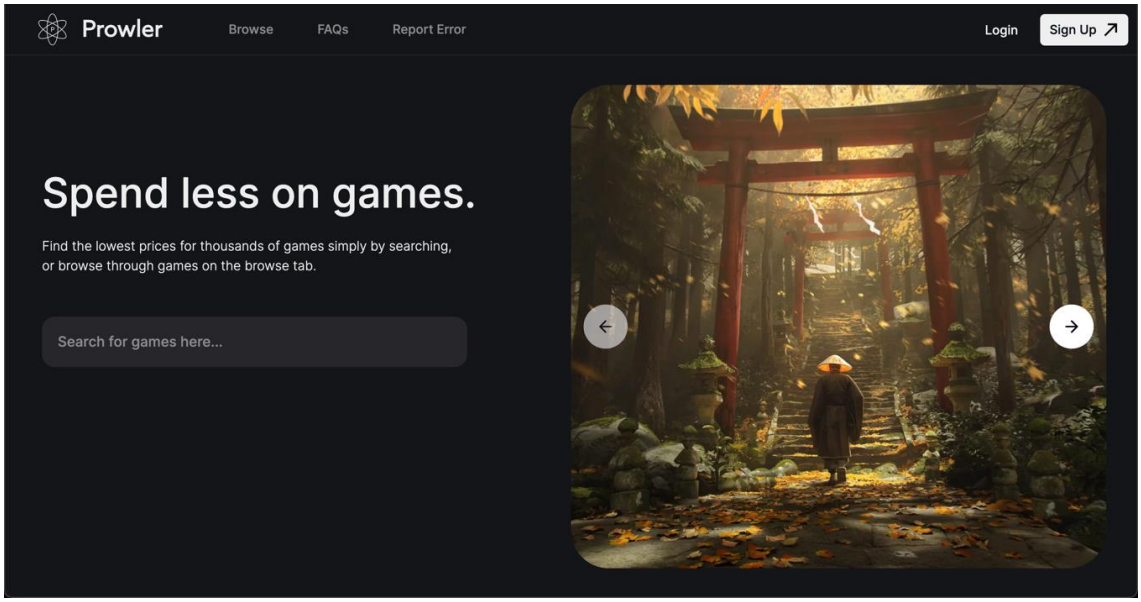
[16] https://www.embla-carousel.com/

and gaming websites especially keep it simple. A lot of people recommend both the Roboto font as well as the Inter font. I downloaded both for now and decided I would decide which font I should use later when I had some more content to view by receiving user feedback and acting accordingly.

I also changed some of the colours. I found I still didn't like the colour I was using so I decided to ask for more stakeholder feedback from my peers. I choose out a darker grey hex which was similar to shadcn/ui's primary colour, and then asked my peers whether they preferred the dark grey or the dark blue primary. Dark colours are certainly the best choice for my site as almost all gaming-related sites use dark hues due to people's subconscious association between gaming and dark colours. I asked four of my peers and ¾ of them agreed that the dark grey hex was made the site look more professional and sleek, so I stuck with that. I also changed the sign up button's colour to my accent colour of white. #EFEFEF would still be the main accent colour of my site which I would use for text, special buttons etc. and I decided that the sign up button, whilst it should still stand out, should also follow the page's colour scheme so it became white. I decided I would keep the navbars links to a slightly darker hex of grey and then make them white upon hover, or when the user is on the link's page (eg. browse is white when on browse page) for visibility of system status.

Main Section of Home Page

Next, I began working on the main section of my home page: the search area on the left. As per my designing, I was going to have a large attention grabbing slogan which would likely be the first thing users looked at, so I made this text 'Spend less on games.' With this, users could immediately recognise the purpose of my site. I also added some less important text below this explaining the purpose in more detail. Underneath both of these I added the search bar, since this was likely what most users would be visiting the site for in the first place. I also added some placeholder text 'Search for a game here…' to let users know that it was a search bar. For my fonts on this page, I was using Inter at this point and testing out different weights for visual hierarchy purposes (eg. bolder for titles and lighter for body text). In order to align the carousel to the right side of the screen, I was just setting the whole page (below the navbar section) to a flexbox, and then applying justify-end to the carousel's parent div.

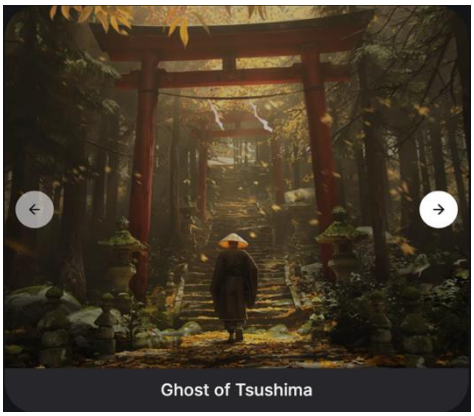


Adding Hover Effect to Carousel Images

Next, I wanted to implement a feature where if the user hovered over the current image in the carousel, then not only would the carousel stop auto scrolling, but a small box would also

slide up at the bottom revealing the game's name and prices. The user should also be able to click on the carousel at this point and be taken to that game's respective page for more detail.

To create sliding animations, I used Tailwind's config file to define custom keyframes. I added two new animations, slideup and slidedown, under the keyframes section. The slideup keyframe will start with the element at 'translateY(100%)' (one height of the element below its position), and ends with the element in its intended position and slidedown is vice versa. Then, I can use the config file to define the animations for these keyframes such as slidedown: 'slidedown 0.2s ease-out forwards', and then finally I am able to use these animation presets on any page. I used this method any time I wanted a sliding animation on my site.



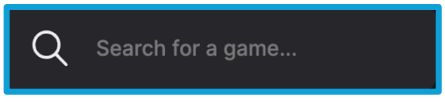
Carousel section after being hovered

Back on the home page, I created a new variable named isHovered and used the onMouseEnter and onMouseLeave functions on the entire carousel to toggle this variable to true or false (true for onMouseEnter, false for onMouseLeave). Then, I added an absolute box at the bottom of the carousel and used JavaScript to check at all times whether isHovered was true. If it was, then the box would have the animate-slideup property which I defined above, else it would have the animate-slidedown property. The final result was when the user hovered the carousel, this box would slide up (I ensured that the box's z-index was lower than the content below it so it was hidden whilst sliding), and then slide down when the user wasn't hovering it. I also added the current game's name to this box by creating another array of each game's name (the indexes were aligned) and then using {carouselNames[index]} to retrieve the correct name for the current carousel item. However, I chose not to include prices in this box because doing so would require a database query, and I wanted to keep my home page static (game's names and links wouldn't require constant updating like price so I could manually add them). Database queries can increase loading times, and since the price data would be the only reason for such a query, I felt that the additional loading time wasn't justified for this extra detail.

To define the isHovered variable, I used React's useState hook. This hook allows functional components to create and manage state variables, enabling them to track user interactions and update the UI accordingly. This basically means that the website will update autonomously based on user inputs or any data changes (providing these actually change the website's content). This feature of React is hugely beneficial for me specifically because my filter system will be changing data and taking user inputs, however with useState I don't need to worry about manually updating the UI every time.

Finishing Touches and Improvements

Next, I wanted to add a magnifying glass icon to the search bar because it wasn't immediately clear what the search box's function was to a new user. On top of this, an icon would better adhere to recognition rather than recall as well as match between system and real world. I decided that I wanted the search icon on the left side of the search box because this was industry standard for more modern websites so I could adhere to consistency and standards. This would turn out to be a small challenge as it was tricky to incorporate a non-clickable icon into a search input since nothing can go inside a <input> tag. I fixed this by making one overall div which is a flex, then adding the search icon and search input separately inside this parent div with relative positioning.



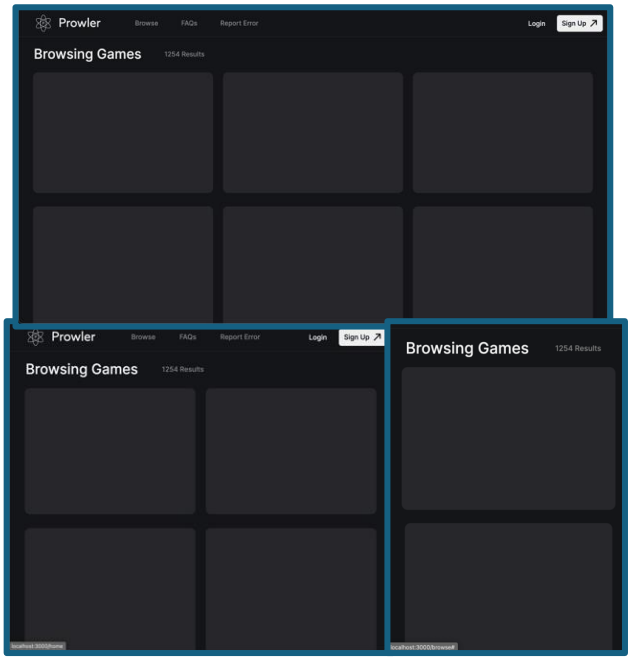
```
// Carousel mapping dictionary
const carouselData = [
  {
    name: "Ghost of Tsushima",
    image: "tsushima.jpg",
    link: "/game?id=1033",
  },
  {
    name: "Sea of Thieves",
    image: "seaofthieves.jpg",
    link: "/game?id=1048",
  },
  {
```

```
{/* Mapping the carousel data object */}
{carouselData.map((item, index) => (
  <CarouselItem key={index} className="basis-1/2 small:basis-1/3">
    <Card className="bg-transparent">
      <CardContent className="flex p-0">
        {/* Conditional rendering: show image with link if available, otherwise show a placeholder */}
        {item.image ? (
          <a
            className="relative overflow-hidden rounded-[18px]"
            href={item.link}
          >
            {/* Display the image with hover effect */}
            <img
              src={`/images/${item.image}`}
              alt={item.name}
              className="h-auto hover:brightness-80 transition-all duration-200"
            />
            {/* Animated text overlay that slides up on hover */}
            <div
              className={`flex flex-col items-center absolute bottom-0 w-full h-[50px]
              bg-lightmidnight z-10 transition-all duration-200 justify-center
              ${isHovered[index] ? 'animate-slideup' : 'animate-slidedown'}`}>
              <p className="font-inter small:text-2xl text-offwhite mb-1">{item.name}</p>
            </div>
          </a>
        ) : (
          // Placeholder for cases of missing images
          <div className="w-full h-full bg-gray-200 rounded-4xl" />
        )}
      </CardContent>
    </Card>
  </CarouselItem>
))}
```

The final change I made to my home page for now was modifying how I was handling the arrays for the carousel. Currently, I was using two separate arrays, one for the games' names and one for the image files. I was about to add a third array for the games' links however this seemed inefficient and was making my codebase messy and harder to maintain. For this reason, I switched to using an object (dictionary) and assigning each game's name with its image and link (added later in development after game page). Then, I simply swapped a couple variable names as needed to streamline the code and enhance readability, resulting in a more organized and efficient structure for managing the carousel data. The full code section for the carousel after I integrated the mapping of the carouselData object is shown above.

Browse Page Development

I proceeded by beginning development on the browse page. My first steps were simply to design a rough layout of the page with static data following the design shown on Adobe XD. Then, once I had laid the groundwork, I would implement the database connection and make the data dynamic. My first version simply consisted of the 'Browsing Games' text, the 'X Results', and a blank grid for the games. I opted for a three-column grid layout to display the games, aiming for a clean and organized presentation that also made responsiveness simple. Tailwind CSS provides an easy way to adjust styles based on screen size using its responsive prefixes. For example, I used a tablet breakpoint to adjust the layout of my games grid. I set tablet:grid-cols-2 on the grid container, which means that when the screen width is 700px or larger, the grid will have two columns instead of one. As the screen size increases further, I could use additional classes like lg:grid-cols-3 to add a third column. This approach ensures that the layout adjusts smoothly as the screen size changes. Additionally, I set the grid container's width to w-full with some padding on the sides instead of a fixed pixel value. This made sure that the grid would automatically adjust to fit the full width of the screen, and since using grid-cols-3 will automatically make each item's width 1/3 of the grid's width, it was fully responsive already.



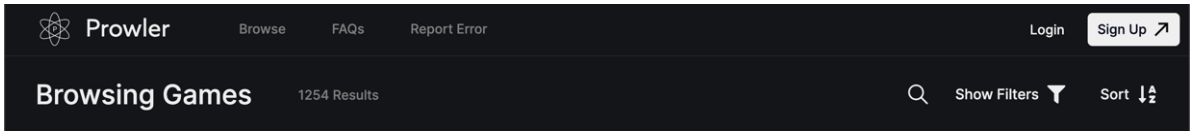
3 different screen widths for browse page demonstrating

Filtering and Sorting Buttons

Next, I created the dynamic search bar, the filter button, and the sort button (without functionality for now) like in the design. I was going to use icons for each of these buttons to improve the user interface as these would address recognition rather than recall and match between system and real world. The search bar icon would be a magnifying glass, which expands into an input field when clicked, allowing users to type in their search queries directly. For the filter button, I planned to use a funnel icon to represent filtering options, and for the sort button, an arrow icon with A to Z next to it. For these icons, I was going to be using React icons [17]which is a website which provides a huge range of icons free to use for developers. They are also easy to use as all you need to do is import them at the start of the file and then you can use them like a regular HTML tag. Another benefit of using icons from React icons instead of images is that you can easily customise their colours, dimensions and even how thick they are. I would end up using this website for just about every single icon I used as they have a huge catalogue and all of their icons were based off industry standards (eg. searching filter would return funnels and cogs).

```
import { IoIosSearch } from "react-icons/io";
import { FaSteam } from "react-icons/fa";
import { SiEpicgames } from "react-icons/si";
import { FaSortAlphaDown } from "react-icons/fa";
import { FaFilter } from "react-icons/fa";
```

```
<FaFilter className="text-offwhite h-5.5 w-auto"/>
```

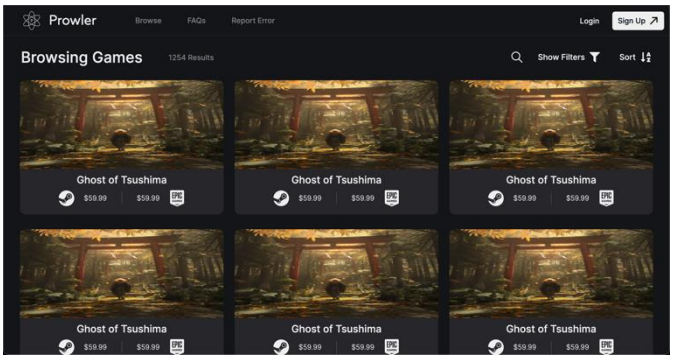


Implementing Static Game Cards

I also added some placeholder static info for games to the page for now. By doing this, I could do all of the styling for cards now and when it came to actually adding the dynamic data, I simply had to change a couple of text fields. However, if I simply created a div for each card in the grid, my codebase would become very messy, repetitive, and hard to maintain, so I followed a similar idea to what I did to fix this problem with the carousel data. I again utilised JavaScript's .map function to dynamically generate the same card (placeholder so same data for each card) nine times. I used 'Array.from({ length: cardCount }).map(...)' to generate a number of cards (cardCount's value) where each card used the same structure but is able to be customized with dynamic data if needed. This approach also allowed me to add or remove cards with the cardCount variable which was useful for testing. Each card includes a placeholder image and static text like "Ghost of Tsushima" and prices, so when I connect it to dynamic data later, I'll only need to update the image source and replace static data with dynamic (eg. $59.99 replaced by price variable). The result of these changes on each card is shown to the right.

```
<div className="flex justify-center w-full">
  <div className="grid gap-x-6 gap-y-8 grid-cols-1 tablet:grid-cols-2 lg:grid-cols-3 mt-1 pb-8 w-full px-9">
    {Array.from({ length: cardCount }).map((_, index) => (
      <a href="#">
        <div key={index} className="h-70 w-card bg-lightmidnight rounded-xl hover:scale-103 transitic
          <Image
            src="/images/tsushima.jpg"
            alt="Game image"
            width={10000}
            height={10000}
            className="w-full h-cardimage rounded-t-xl"
          />
          <div className="flex flex-col items-center">
          <p className="font-inter text-2xl text-offwhite mt-top">Ghost of Tsushima</p>
          <div className="flex mt-1">
            <FaSteam className="text-offwhite mt-0.5 mr-5 h-logos w-auto"/>
            <p className="font-interlight text-prices text-offwhite mt-2">$59.99</p>
            <div className="bg-grey h-line w-0.2 mt-2.3 mx-8"/>
            <p className="font-interlight text-prices text-offwhite mt-2">$59.99</p>
            <SiEpicgames className="text-offwhite mt-0.5 ml-5 h-logos w-auto"/>
          </div>
          </div>
        </div>
      </a>
    ))}
  </div>
</div>
```

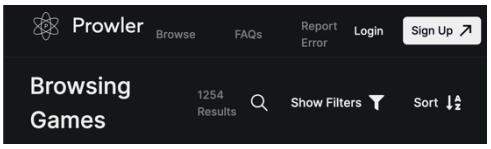
Code for each game's card

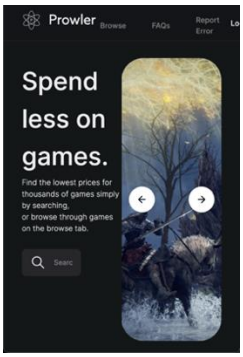


[17] https://react-icons.github.io/react-icons/

Incorporating User Feedback and Enhancing Usability

At this point I decided to get another round of stakeholder feedback. I asked some of my friends who weren't in my digital technology class so that I could get a wider range of feedback, specifically from people who weren't as acquainted with my site. I showed the browse page to five people. One of the main suggestions was to make the navbar stay at the top of your screen as you scrolled, aka a sticky navbar. I thought this was a great idea because it makes the site still easy to navigate even for users who may have scrolled far down on the browse page. This was very simple to implement as I could just add the 'sticky' property to the entire navbar, ensured that the navbar had the highest z-index of all elements on the website, and I also made it have a slight blurry transparent effect to give it a modern look whilst maintaining focus on the content below.





The final piece of feedback I received was something I was aware of but putting off until this point – the poor responsiveness of my site which is showcased to the right. Thus, I decided I would work on responsiveness before moving on. Responsiveness is crucial for my website because many users will want to quickly check game prices on their mobile devices while on the go. A well-optimized responsive design ensures that they can easily navigate, browse, and access the information they need, enhancing their overall experience and encouraging them to return.

It's important to note that at this point in my development process, I was still using w-full for the overall containers for each of my pages, which is bad practice for reasons I will explain later (I end up explaining this on page 58). Despite this, at this point I was still figuring out how I wanted my pages to look when the window size got thinner and how to put this into practice. I began using the CSS unit 'em' instead of pixels because I did some research on how modern websites do responsive design, and found that majority of them used 'em' for sizing. 'em' is a relative measurement, and its size depends on its 'parent' element, so if the text size is 16px, then a margin of 1em will be 16px. It's widely agreed upon that 'em' is superior than pixels because it allows for more flexible and scalable designs, because the sizes will automatically adjust to its parent element. This enhances responsiveness as it allows margins and text sizes to adapt more fluently across different screen sizes. For these reasons, I adapted most of my margins, text sizes, and widths/heights to em values.

Adapting for Mobile Responsiveness

Next, I needed to adapt the navbar, home page, and browse page for smaller screen sizes, as they were frankly a mess as shown above. Fortunately, I already had ideas for fixes due to designing mobile views for these pages on Adobe XD. For the navbar, I was going to use Tailwind's media queries to conditionally remove the links and login/sign up buttons, and instead render one hamburger menu which stores all of these. I was going to do the same thing for the browse page also by adding one 'Filter' button on smaller screens. Lastly, for the home page, I would switch from a 2 column layout to a single column when the screen was sufficiently small. I did however decide that I would not code the actual responsive menus (the menus which show up when you press the hamburger menu/filter button) because I hadn't even coded the regular filter menu yet.

For the browse filter button, I created a new custom screen size named browsewidth, which was 820px wide. I chose this width because it was when the ‘x results’ and search icon were about to touch, so implementing the filter button at this point would avoid this. Then, all I needed to do was hide the parent div of the search bar, show filters button and sort button by default, and use browsewidth:flex so that they only were displayed above this width. Lastly, I just had to create the filter button and do the vice versa by using browsewidth:hidden. Before and after photos of this are shown below the next paragraph with the navbar’s before and after photos.

```
<div className="hidden browsewidth:flex ml-auto mr-6">
    <a href="#"
    className="flex hover:bg-lightmidnight transition-colors duration-200
    h-10 w-10 rounded-lg justify-center items-center mr-2.5">
        <IoIosSearch className="text-offwhite h-8 w-auto"/>
    </a>
```

Parent div of the search icon, show filter button and sort button

```
<div className="flex browsewidth:hidden ml-auto mr-3">
    <button
    className="flex hover:bg-lightmidnight transition-colors duration-200
    h-10 w-26 rounded-lg justify-center items-center mr-3">
        <p className="text-offwhite font-inter text-filter mr-2.5">Filter</p>
        <FaFilter className="text-offwhite h-5.5 w-auto"/>
    </button>
</div>
```

New filter button code

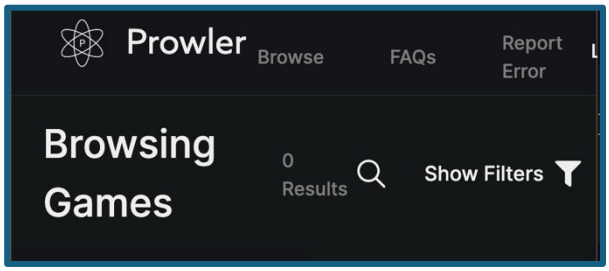
I then proceeded to follow an identical technique using hidden and block for the navbar responsiveness with the screen width preset ‘navwidth’. I found the width where the content needed to be less wide, and then used navwidth:hidden in the same way I did above to conditionally render the hamburger menu at small widths less than navwidth, and render all the other buttons and links at widths beyond navwidth. A before and after photo of this is shown below on the next page.

```
<div className="hidden navwidth:block">
  <ul className="flex items-center gap-6 ml-5">
    <a className="font-inter text-base text-grey hover:text-offwhite
t transition-colors duration-200" href="../browse">Browse</a>
    <a className="font-inter text-base text-grey hover:text-offwhite
t transition-colors duration-200" href="#">FAQs</a>
    <a className="font-inter text-base text-grey hover:text-offwhite
t transition-colors duration-200" href="#">Report Error</a>
  </ul>
</div>
```

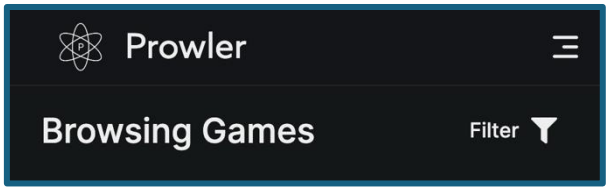
Parent div of links

```
<CgMenuRightAlt className="text-offwhite h-8 w-8 navwidth:hidden"/>
```

Hamburger menu

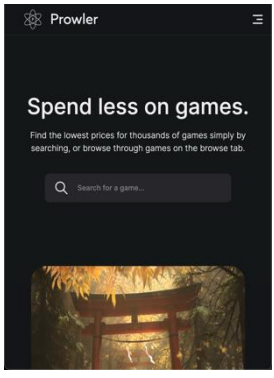


Mobile view before the above changes



Mobile view after

The responsiveness of the homepage is achieved through a combination of flexible layout management and media queries. The page is setup as a dual-column layout for larger screens, where the left section (section with search bar) maintains a minimum width that prevents it from shrinking below half the screen width, ensuring readability and aesthetic balance. This is controlled by setting minWidth of the left section based on the initial screen width. The carousel on the right side however adjusts its width dynamically, allowing it to resize with the screen. At a breakpoint of 1200 pixels, the layout transitions to a single-column format, allowing for a compact but still usable version on smaller screens. Additionally, I used Tailwind’s media queries to centre the text and search bar as it allows for consistent spacing and alignment and it’s standard for single column layouts to use centre alignment.



Mobile view of home page

```
<div className="homewidth:flex homewidth:justify-between h-[calc(100vh-65px)]">
  <div className={`homewidth:w-half homewidth:items-start homewidth:self-center homewidth:mt-0 mt-28 flex flex-col items-center mx-4`}
  style={screenWidth >= homewidth ? { minWidth: `${initialScreenWidth * 0.5}px` } :{}}>
```

Code for parent div of whole home page, and then parent div of left section. Any properties preceded with homewidth: means they are only applied in the two-column layout on large screens

Website Database Connection

At this point I decided that I had a solid blueprint for my website and was ready to implement the database connection. This meant I would need to first connect with my local database on phpMyAdmin, get the data to the browse page, and then appropriately format it. I had little to no experience with database interaction in React so I decided I would initially follow a tutorial on this subject to build some foundational knowledge. I chose this YouTube tutorial[18]. Following the steps, I created a config folder in the Next.js folder (website folder) and then a db.js file inside of this config folder. Then, to firstly establish the database connection, I used the mysql.createConnection function and copied and pasted over the appropriate environment variables for my database. Lastly, I used db.connect as a testing connection, and then checked for any errors and returned if the connection was successful. Upon testing with node.js, I was able to successfully connect to my database shown to the right.

```
+const mysql = require("mysql");
+
+const db = mysql.createConnection({
+    host: "localhost",
+    user: "root",
+    password: "2WS8YL8hqh988NxaPVP2iufPv",
+    database: "prowler_games",
+});
+
+db.connect(err => {
+    if (err) {
+        console.error("Error connecting to MySQL database:", err);
+    } else {
+        console.log("Connected to MySQL database");
+    }
+});
+
+export{db}
```

```
kobe@Kobes-MacBook-Air Prowler % cd ./website
kobe@Kobes-MacBook-Air website % cd ./config
kobe@Kobes-MacBook-Air config % node db.js
Connected to MySQL database
```

The next step in the tutorial was to create a data.js file in the same config folder. This file was going to be actually querying the database which db.js has established a connection with. In this file, I imported express and cors modules to create a server that can handle HTTP requests and enable cross-origin resource sharing (CORS). Additionally, I imported the db object to interact with my database. After setting up the Express app and enabling CORS for cross-origin requests, I created a route for GET /games. This route executes an SQL query to retrieve all data from the games table in the database. If an error occurs during the query it will logs the error and returns a server error message. Otherwise, it sends back the data from the SQL query in JSON and the server listens on port 8081. The end result of this is all the entries on the database will be stored via JSON on localhost:8081/games (when the server is running) as shown below.

```
+import express from "express";
+import db from "./db.js";
+import cors from "cors";
+
+const app = express();
+
+app.use(cors());
+
+app.get("/games", (req, res) => {
+    const sql = "SELECT * FROM games";
+    db.query(sql, (err, data) => {
+        if (err) {
+            console.error("Error ", err);
+            return res.status(500).json({ error: "Server Error" });
+        }
+        return res.json(data);
+    });
+});
+
+app.listen(8081, () => {
+    console.log("Listening via port 8081...");
+});
```


```
    "game_id": 682,
    "name": "Fallout 4",
    "reviews": "Very Positive (1,685)",
    "steam_on_sale": "0",
    "steam_normal_price": "N/A",
    "epic_on_sale": "-1",
    "epic_normal_price": "-1",
    "developer": "Bethesda Game Studios",
    "publisher": "{\"Bethesda Softworks\"}",
    "short_desc": "Bethesda Game Studios, the award-winning creators of Fallout 3 and The Elder Scrolls V: Skyrim, welcome you to the world of Fallout 4 – their most ambitious game ever, and the next generation of open-world gaming.",
    "long_desc": "About This Game\r\n\t\t\t\t\t\t\t\t\tBethesda Game Studios, the award-winning creators of Fallout 3 and The Elder Scrolls V: Skyrim, welcome you to the world of Fallout 4 – their most ambitious game ever, and the next generation of open-world gaming.As the sole survivor of Vault 111, you enter a world destroyed by nuclear war. Every second is a fight for survival, and every choice is yours. Only you can rebuild and determine the fate of the Wasteland. Welcome home.Key Features:Freedom and Liberty!Do whatever you want in a massive open world with hundreds of locations, characters, and quests. Join multiple factions vying for power or go it alone, the choices are all yours.You're S.P.E.C.I.A.L!Be whoever you want with the S.P.E.C.I.A.L. character system. From a Power Armored soldier to the charismatic smooth talker, you can choose from hundreds of Perks and develop your own playstyle. Super Deluxe Pixels!An all-new next generation graphics and lighting engine brings to life the world of Fallout like never before. From the blasted forests of the Commonwealth to the ruins of Boston, every location is packed with dynamic detail. Violence and V.A.T.S.!Intense first or third person combat can also be slowed down with the new dynamic Vault-Tec Assisted Targeting System (V.A.T.S) that lets you choose your attacks and enjoy cinematic carnage.Collect and Build!Collect, upgrade, and build thousands of items in the most advanced crafting system ever. Weapons, armor, chemicals, and food are just the beginning – you can even build and manage entire settlements.",
    "banner": "https://shared.akamai.steamstatic.com/store_item_assets/steam/apps/377160/header.jpg?t=1726758475",
    "images": "{https://shared.akamai.steamstatic.com/store_item_assets/steam/apps/377160/ss_f7861bd71e6c0c218d8ff69fb1c626aec0d187cf.1920x1080.jpg?t=1726758475,https://shared.akamai.steamstatic.com/store_item_assets/steam/apps/377160/ss_f7861bd71e6c0c218d8ff69fb1c626aec0d187cf.1920x1080.jpg?
```


Implementing Axios Request

Next, I needed to retrieve this information on the browse page and appropriately format it. For this, I was going to use Axios, which is a HTTP client for JavaScript that allows you to make requests to external pages and handle responses in a more convenient way than using the native fetch function. This meant I could easily make a request at localhost:8081/games and retrieve all the data on the browse page.

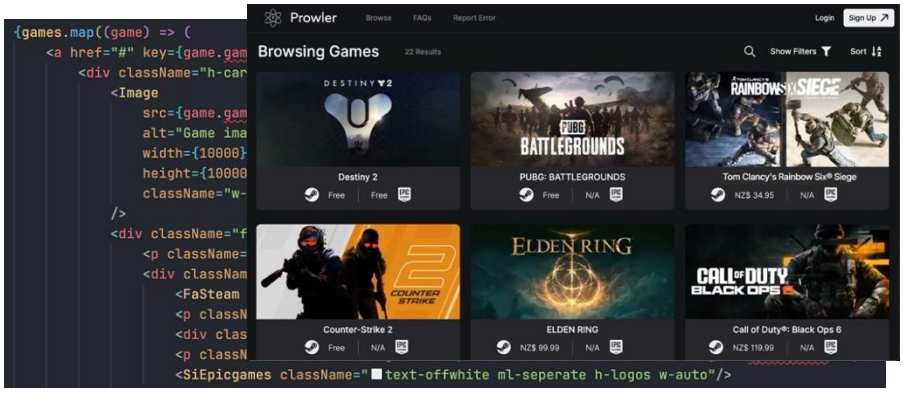
[18] https://www.youtube.com/watch?v=Q3ixb1w-QaY&ab_channel=CodeWithYousaf

```
const [games, setGames] = useState([]);

useEffect(() => {
    axios.get("http://localhost:8081/games")
        .then(response => {
            setGames(response.data);
        })
        .catch(error => {
            console.error("There was an error fetching the game's information: ", error);
        });
}, []);
```

When going about my Axios request, I decided to use React's useEffect hook. This hook essentially will run the snippet of code inside of it in specific circumstances. It will run immediately after the first render (when the page loads), and then it will only run again if one of the variables (dependencies) in the dependency array updates at any point. This is especially beneficial for my website because, although the dependency array is empty now, my filter system means that the page will more than likely need to request from the database multiple times as the user customises their filter selections. Thus, if I include the variables for filter selections in the dependency array for my database request, then it will run the request each time the user updates a filter completely autonomously. Within the useEffect hook, I used an axios.get function and passed it the appropriate link of localhost:8081/games, then set a variable named games to the response data returned. I also integrated some error handling which logged the specific error raised for future proofing. Thus, the end result was that the games variable was now an array and each item in this massive array was a game from the database.

Mapping Games Dynamically in Cards

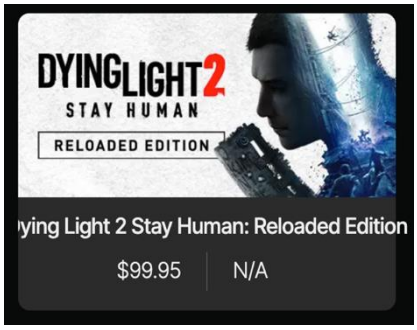
Now that I had this array, I needed to appropiately map each game to a card. For this, I followed a similar idea to what I did for the placeholder cards, and mapped the games variable. I named the item for the current iteration of the mapping as 'game' (similar to for game in games loop). Then, because I had already prepared the cards for becoming dynamic when I made the placeholders, I simply needed to change the text fields like changing the title from 'Ghost of Tsushima' to 'game.game_name' (game is the current item and game_name is the column in the database). I followed this process for the image source, name, and prices. I also didn't need to specify a number of cards because the mapping function would iterate through each item in the games array meaning every game would be displayed automatically. After finishing this up, my browse page was displaying every game from the database as shown to the right. Currently, these were being displayed in an infinitely scrolling manner, however I would eventually implement pagination to combat poor performance with this method.



Code section for each card at this point

Handling Abnormally Long Game Names

However, upon scrolling through the browse page, I noticed that a few games which had especially long names were causing some issues. Their names were so long that the text ended up exceeding the width of the card itself. This was being caused because I had set the text-wrap limit to one (only one line of text height) because if only some games wrapped and went to two lines then it could cause many unwanted nusiances with styling. My solution to this problem was to shorten the names of games which exceeded a certain character limit, and then if the user was interested in the game's full name then they could hover it and the full name would appear in a small label above.



Example of the games I'm referring to

Firstly, I used JavaScript to check the character length of the game's name, and if it was greater than 30, then I would assign the game's name to a substring of the first 30 characters

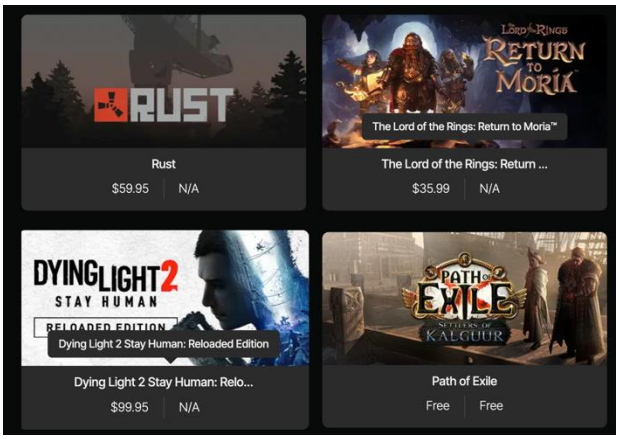
plus '…' at the end, as well as set isLong to one (binary so one for too long and zero for not). I came to the limit of 30 after testing different character limits, I also had to test that this limit wouldn't exceed the width of the card when the screen and thus cards were more narrow. Code for this check is below.

```
// Shortening games names which are too long
if (gameName.length > 30) gameName = gameName.substring(0, 30) + '...', isLong = 1;
```

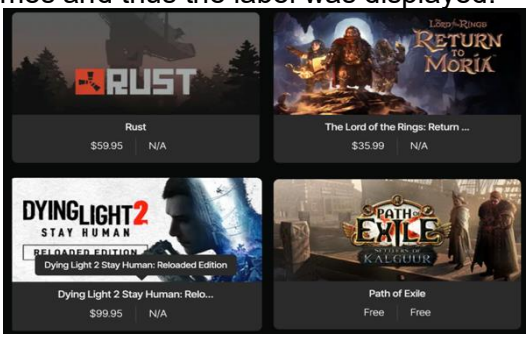
Next, I had to code the small label which appeared upon hovering a game's name which was shortened. This label would simply contain the game's full name in smaller font size. Thus, the two conditions for this label to appear were that the game's name was too long (which I could track with the isLong variable) and that the name's text was being hovered. Therefore, I added a variable named hoveredGame and set it to true when defining. Then, by using onMouseEnter, I could set this variable to true when the user hovered the game's name. Finally, I could check that both isLong and hoveredGame were true (JavaScript takes 1 as true) and set the label's opacity to 100% if so, or to 0% if either was false. Upon testing, the label did appear, however it actually appeared for every single game which had a shortened name. This is due to the hoveredGame variable being set to true universally when the mouse enters any game name element, rather than being uniquely tied to the specific game being hovered, and since all other games with shortened names also had isLong = 1, both conditions were met for all of these games and thus the label was displayed. To fix this issue, I changed the variable to a number type named hoveredGameId, and then upon hovering the game's name, I set this variable to the id of this specific game. I also made the variable null when the mouse left. Then, in the if condition for whether the label should be displayed, instead of checking for true or false, I checked if the hoveredGameId was equal to the game's id. This condition was thus only met for the game whose name was being hovered as intended.

```
<div className={`absolute flex py-2 px-4 bg-lightmidnight items-center rounded-popup
    content-center transition-all duration-150 mb-3 triangle-div text-center
    // Conditionally rendering a small popup for long game names
    ${isLong && hoveredGameId ? 'opacity-100' : 'opacity-0'}`}>
    <p className="text-offwhite font-inter">{game.name}</p>
</div>
```

Label's code section, the if statement is the ${isLong… line



Two games with shortened names both with labels even though only one is being hovered



Label now only showing for the game being hovered

```
onMouseEnter={() => setHoveredGameId(game.game_id)}
onMouseLeave={() => setHoveredGameId(null)}>
{gameName}
```

Game's name section, onMouseEnter sets variable to game id, onMouseLeave sets it to null

```
// Conditionally rendering a small popup for long game names
${isLong && hoveredGameId === game.game_id ? 'opacity-100' : 'opacity-0'}`}>
```

Conditional if statement on label, only passes when isLong = 1 and for the current game being hovered. Opacity used here as using hidden doesn't allow for smooth animations

Development of Filter Section

My next major task was to finish off the browse page. For this, I needed to add a regular filter section, responsive filter menu (for small screens), sorting menu, dynamic searchbar, and the backend functionality for these features. I started by developing the filter section, taking inspiration from Nike's design. This design was a bar which slid out from the left side which you could also toggle to hide. I liked this feature because it ensured users could easily customize their browsing experience while keeping the interface clean and user-friendly.

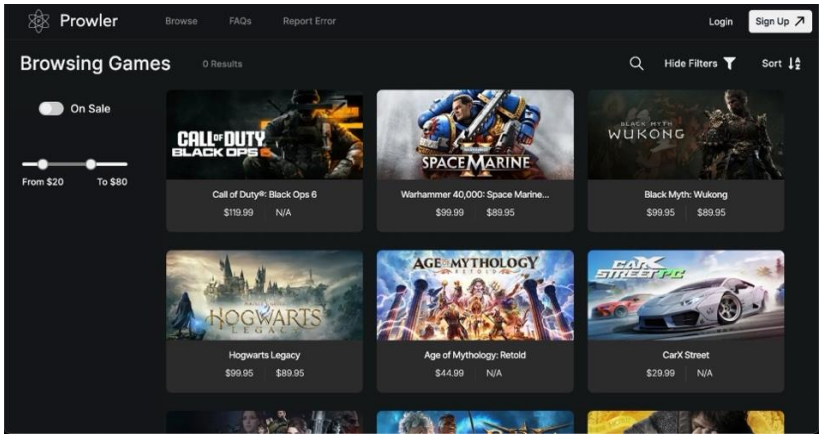
```
<button
className="flex ☐hover:bg-lightmidnight transition-colors duration-200
h-10 w-40 rounded-md justify-center items-center mr-3"
onClick={() => { setShowFilter(!showFilter); setClickedButton(true); }}>
    <p className="■text-offwhite font-inter text-filter mr-2.5">
        {showFilter ? 'Hide' : 'Show'} Filters
    </p>
    <FaFilter className="■text-offwhite h-5.5 w-auto"/>
</button>
```

I started by adding a new variable named showFilter which was set to false by default and would track whether the filter bar was displayed or hidden. On the 'Show Filters' button I had already made, I used JavaScript's onClick function to set showFilter to the opposite of its current value, essentially meaning the button acted as a toggle for the filter section. I also made the button's text dependant on showFilter's value, so that when it was true, the button read 'Hide Filters' (since the filter bar would be out) and vice versa when it was false.

Next, I created a parent div for the filter section which had a set width of 250px and height of 100%. I then used the same method I explained for the carousel popup to animate the filter bar sliding to the right, out of the left side of the screen. I ensured that I applied overflow-hidden to the parent class of the filter section because if I didn't then the section would be visible to the side of the main content (where the padding is) since I was using transform for the smooth animation (using hidden doesn't allow for smooth animations as it instantly disappears). Also, I didn't need to worry about the game cards as I had set the parent width of them to w-full which meant that they would automatically resize to accommodate the filter section without issue.

Contents of Filter Menu

I began working on the filter menu contents next and I was going to have 3 filtering categories, as per my design, them being on sale, price and genres. I downloaded and imported a switch component from shadcn to use for my 'On Sale' toggle and did the same for a slider for the price slider. Although I said I was going to use checkboxes earlier, I decided to experiment with a slider and allow user feedback to decide if they preferred the checkboxes or slider. This was quite fast due to shadcn's quick and easy process to use their components. After importing the components, I only needed to change some basic styling before they were finished. This is another huge upside of using pre-built components as I can save large chunks of development time. They weren't functional yet as I decided I wanted to do all of the filtering and sorting backend processes after I had finished designing the frontend for them all.



Browse page with filter bar extended on the

Subsequently, I developed the genres section with checkboxes from shadcn also. This was very handy as the checkboxes came with proper label assignment functionality (using htmlFor and id) so even if a user clicked the text rather than the checkbox, it would still register as clicking the box. When adding the genres checkboxes, I created an array named genres and added around 12 genres to it. Then, as I have done previously, I used the .map(genre) function on this array and thus dynamically coded each checkbox and label. This makes my codebase far more concise and future proofed, as if I wanted to delete, update, or add a genre, I can do so simply in the array rather than editing whole sections of code.

```
{genres.map((genre, index) => {
    const isExtraGenre = index >= 5;
    if (!isExtraGenre || (isExtraGenre && showExtraGenres)) {
        return (
            <div key={genre} className="flex items-center mr-5">
                <Checkbox
                    id={`genre-${index}`}
                />
                <Label
                    htmlFor={`genre-${index}`}
                    className="■text-offwhite font-interlight ml-3
                >
                    {genre}
                </Label>
            </div>
        );
    }
    return null;
})}
```

Genres filtering section

'Show More/Less' button code

I ran into an issue where the number of genres being displayed was too large - around 12 - which caused usability problems as users would need to scroll through a long list of options, making the interface feel cluttered and less intuitive. To solve this, I cut down the normally visible genres to the top five, and added a 'Show More' button which when pressed would show all genres and also have a 'Show Less' button. I added a showExtraGenres variable which would be toggled by the 'Show …' button (pressing show more toggles it to true and vice versa). In order to implement this in a fluid manner with only one genres array, I added an index parameter in the mapping function. Then, the program would check the genre's index in the array, and only allow genres of index five or less to be displayed by default. If the genre was an extra (index is >5) and showExtraGenres is true, it renders that game, otherwise it returns null to skip this rendering (shown in image above).

First Widescale Stakeholder Feedback

Since it was Term 2 Week 7 and I felt it was an appropriate time, I decided I would receive my first set of widescale stakeholder feedback. In order to do this, I was going to deploy a live testing version of my website and send out a Microsoft form for stakeholders to fill out. Using a live testing server allows people to access and test the website from their own devices, providing more realistic feedback on user experience, responsiveness, and compatibility. It also enables me to gather feedback from a wider audience at any time they want, rather than being limited to those who can physically view my laptop.

I also decided that if I was going to deploy a live testing server, I should move my database from being local (phpMyAdmin) to on the cloud as they offer improved accessibility, scalability, and reliability. A cloud database can be accessed from anywhere, making it easier for my application to handle user requests in real-time, and benefits from automated backups and security features from the provider, ensuring better data management and uptime. Additionally, this feature also ensures that my live testing server mirrors the production setup more closely, which makes it easier to test user interactions as well as ensuring that data is synced.

Moving Database to the Cloud

When choosing a cloud database host, most providers offer very similar functionality, making them nearly identical in practical terms - features like scalability, data security, and support for various database types are often standard. I initially looked at the most popular providers such as Microsoft Azure or Oracle, which are known for their robust infrastructure and used by many enterprises. Whilst these platforms do come with some form of free trial or free plan, I found that their functionality was limited unless I paid, which wasn't ideal for minimizing costs. Vercel, on the other hand, offers a generous free plan of 256MB storage size and 60 hours of computing time (per month for both) which would be more than enough for my project. Additionally, Vercel's setup would no doubt be seamless with my Next.js project since they developed the framework. Lastly, Vercel also includes the ability to deploy live versions of my app without charge meaning I could use this platform for my live testing server also. This meant that all of my project's needs, from hosting to deployment, could be met within Vercel's ecosystem, making it a convenient and cost-effective choice.

Vercel offered four database types: Postgres, KV, Blob, and Edge Config. I decided to go with Postgres because it offers a robust and relational structure which is suited for handling complex queries and joins. On top of this, I had vast experience with SQL databases and only Postgres used SQL for interactions (KV or key value store uses NoSQL on Vercel). I proceeded by following Vercel's Postgres documentation[19], and managed to create an empty

[19] https://vercel.com/docs/storage/vercel-postgres/quickstart

Postgres database. I then added a Vercel 'project' by importing my GitHub repository and connected the project to the database. Subsequently, I created a .env file in my parent folder which contained all of the environment variables for the Postgres database.

Next, I needed to populate my database. Following the documentation, I created my three identical tables (games, game_genre, genres) and ensured to copy over the correct character limits and types so I could have a seamless database transition. In order to populate these tables, I then copied my environment variables over to my scraping programs and tried to run the genre scraper (genres must always be populated first). I quickly received an error which the cause of was because I was still trying to use mysql.connector, even though I had switched to a PostgresSQL type database (phpMyAdmin uses MySQL). Thus, I switched over to the appropriate connector which was named psycopg2 and ran all three of my scraping programs. Upon checking the tables on Vercel's website, the expected data was there meaning I could now work on connecting the new Postgres database to my website also.

```
# connecting to the prowlerdb database
games_db = psycopg2.connect(
    host="ep-fancy-frost-a715bifg-pooler.ap-southeast-2.aws.neon.tech",
    user="default",
    password="180bWTweLUqB",
    database="verceldb"
)
```

Revamping Data Retrieval

Because I was going to be using a live testing server, I needed to slightly alter my method of retrieving data from the database. Before, I was establishing a connection, writing to the database, and then storing the data on port 8081, which was functional for local testing. This is not suitable for a live website however, so I decided I would switch to the Next.js's API routes. These would allow me to create serverless functions for handling requests and make my backend more straightforward by managing database interactions through API endpoints. These endpoints are accessible as they are public APIs however the core backend logic is kept separate and I display all data from my database on the website anyway. This transition would improve the efficiency of my data fetching and make maintaining database interactions more simple.

Upon recommendation from one of my peers who was quite experienced in React, I also decided I would be switching from an express server to Prisma ORM for my database connection and interaction. Prisma ORM (object relational mapping) is a modern database toolkit that simplifies accessing databases through its type-safe and beginner-friendly API, and allows interaction with a database utilising TypeScript or JavaScript instead of needing to write raw SQL (although it has its own specific syntax too). It offers a simplified setup, more readable and intuitive code, and robust security by handling database connections more efficiently than manually opening a client. This switch would make my database connection and interaction more concise and easy to maintain. Lastly, Prisma also sanitizes all SQL queries before running them, meaning my website would be completely secure against SQL injections.

In order to implement these changes, I first deleted the config folder and made an api folder within the project's root folder, along with a route.ts file (all api endpoints must be named route). I also followed Prisma's documentation and created a schema file (contains database structure) as well as a migrations folder for backups of the schema. In the route.ts file, the code initializes a Prisma client and defines an asynchronous function for GET requests. In the function, it tries to fetch all records from the games table with the line 'prisma.games.findMany()' (this is Prisma's 'language' or syntax). If the fetch is successful, then the function returns this data as JSON, otherwise it will return error with the specific message. Due to the file being an

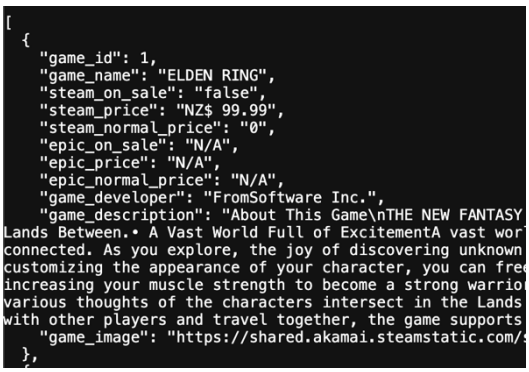
```
import { NextRequest, NextResponse } from 'next/server';
import { PrismaClient } from '@prisma/client';

const prisma = new PrismaClient();

export async function GET(req: NextRequest) {
    try {
        const games = await prisma.games.findMany();
        return NextResponse.json(games);
    } catch (error) {
        console.error(error);
        return NextResponse.error();
```

```
model games {
  game_id           Int       @id @default(autoincrement())
  name              String?   @db.VarChar(200)
  reviews           String?   @db.VarChar(40)
  steam_on_sale     String?   @db.VarChar(40)
  steam_price       Float?
  steam_normal_price String?  @db.VarChar(40)
  steam_link        String?   @db.VarChar(200)
  epic_on_sale      String?   @db.VarChar(40)
  epic_price        Float?
  epic_normal_price String?   @db.VarChar(40)
  epic_link         String?   @db.VarChar(200)
```

Section of Prisma's schema file

API endpoint, visiting /api on the website will then display this JSON successfully, similar to the localhost:8081 page.

Back on the browse page, since the structure of the data is identical, all I needed to edit was changing the axios.get parameter to '/api' rather than 'localhost:8081' and the page was accommodated to the change as the data being retrieved is equal to before. Because it searches for /api from the root directory of the project, this means that the request will work successfully on a live testing server and locally since whatever is before /api doesn't matter (eg. whether the link to the api page is prowler.com/api or prowlergames.com/api, the axios get will still retrieve data successfully). Hence, the data was once again being displayed on the browse page however this time it was coming from my database hosted by Vercel rather than locally.



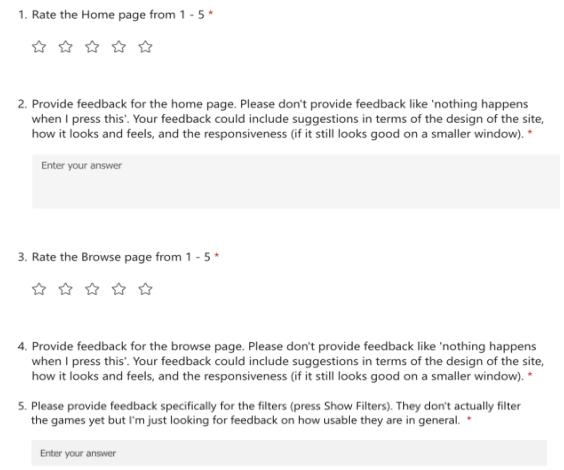
```
[
  {
    "game_id": 1,
    "game_name": "ELDEN RING",
    "steam_on_sale": "false",
    "steam_price": "NZ$ 99.99",
    "steam_normal_price": "0",
    "epic_on_sale": "N/A",
    "epic_price": "N/A",
    "epic_normal_price": "N/A",
    "game_developer": "FromSoftware Inc.",
    "game_description": "About This Game\nTHE NEW FANTASY
Lands Between.• A Vast World Full of ExcitementA vast wor
connected. As you explore, the joy of discovering unknown
customizing the appearance of your character, you can fre
increasing your muscle strength to become a strong warrio
various thoughts of the characters intersect in the Lands
with other players and travel together, the game supports
    "game_image": "https://shared.akamai.steamstatic.com/
  },
```


The API endpoint that the axios get function pointed to. Data being fetched by Prisma from database on Vercel

Receiving Stakeholder Feedback

Next, I needed to deploy a live testing server for feedback from stakeholders. When attempting to deploy my website on Vercel, I repeatedly got errors for TypeScript specific syntax errors such as needing to change NextApiRequest to NextRequest. Eventually, I managed to rid all of these minor errors and was able to deploy the first live testing server[20] for my website.

Subsequently, I began creating the Microsoft form[21] for feedback. I included five questions, shown to the right, which covered feedback for the home page, feedback for the browse page, and feedback specifically for the filter section. I wanted feedback specifically for the filter section because I had just finished it and it was a vital component of the page so I wanted to ensure it was the best it could be. Additionally, I also wanted feedback for the price slider as I mentioned previously so I could finalise my decision between checkboxes or slider. Lastly, I sent out an email to ten stakeholders with the form and the testing server's link (in the form). One thing I did learn from this survey was that people don't like answering survey questions which are too wordy, so I kept this in mind for the future. Overall, I received lots of positive feedback surrounding the design and layout, although I also received ample negative feedback for me to work on.

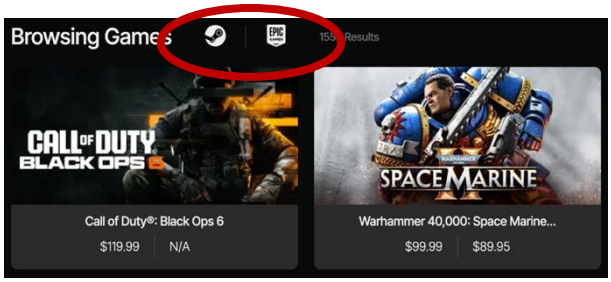


Email sent to stakeholders

Feedback: "At some resolutions the Steam logo goes off the side of the game card (not enough width for the price and the logo)"

[20] https://prowler-mxug9ecj6-kobe-bayliss-projects.vercel.app/home
[21] https://forms.office.com/r/fzE2n5qeVw

To fix this, I decided that I would remove both the Steam and Epic Games logos from each game's card. I felt that having these logos in every card was somewhat redundant as the user would understand that Steam was left and Epic was right after a point, and thus I decided to place the logos at the top of the page as shown. This placement is intuitive enough for users to understand whilst also removing unnecessary clutter on the page. Because of this, the feedback was also addressed as even the longest names would still stay within the card's width now since there were no logos.



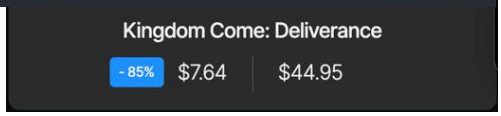
Feedback: "make sure it is displayed when a game is on sale when browsing or searching i.e., 80% off"

Sale Indicator

Upon reading this, I immediately thought it was a great idea and wondered why I didn't add this prior to this point. By having some form of indicator which stands out for games on sale, it would attract the user's attention and make it easier for them to spot the best deals, thus improving my site's user retention as users will want to use my site to find deals. To address this, within each game's card, I added two bright blue boxes, which stand out against the dark background, to the side of the price (where the logos used to be). These boxes conditionally rendered if the respective platform had the game on sale at that moment (eg. if the game had a sale on Steam but not Epic, only the box next to the Steam price renders), which I did by checking the steam_on_sale and epic_on_sale values in the database (1 for on sale, 0 for not). Inside this box, I put the game's discount percentage for the respective platform, which I simply calculated using JavaScript (logic is (price not on sale – price on sale)/price not on sale * 100). Lastly, since this new feature was essentially replacing the platform logos, I would again need to consider these boxes exceeding the card's width at smaller screen widths. To do this I played around with the breakpoints so that the page would go to two card rows sooner when shrinking, and made the text sizes slightly smaller and this fixed the issue. Game with a sale box is shown to the right.

```
// Below I calculate both the epic and steam discount %
if (steamOnSale) {
    let normalPrice1 = steamNormalPrice;
    let salePrice1 = steamPrice;

    if (normalPrice1 > 0) {
        steamDiscount = Math.round(((normalPrice1 - salePrice1) / normalPrice1) * 100);
    }
}
```

Calculating the discount percentage




```
{steamOnSale && (
    <p className="font-inter m
        - {steamDiscount}%
    </p>
)}
```

Code for the blue box with if statement to check that game is on sale

Feedback: "a price slider for filtering video games seems out of place, would expect checkboxes or something else maybe"

Switching Price Filtering to Checkboxes

This was the specific feedback I was looking for as a stakeholder confirmed my hypothesis. Whilst a slider can offer superior customisability when filtering for price, checkboxes have been proven to enable faster and more efficient filtering. Another reason why checkboxes would be a better choice for usability for my specific project is because when people are looking to buy a new game, they are going to be looking for a game with a price within a general range rather than a specific one. For example, if a user comes to my site and wants to buy a cheap game, they would probably prefer to filter by a 'Free to $20' checkbox over manually selecting this range, and the portion of users who do want to select a very specific range would be miniscule. Thus, I decided I would implement checkboxes for my price filtering also. This was very quick because it was essentially identical to the genres checkboxes (since it was only frontend at this point). I also didn't need to worry about any 'Show More' button since there would only be four or five price ranges. The final result after the change is shown to the right.

Filtering Backend Phase

This was all the feedback I deemed worthy of addressing so I promptly moved on to my filtering backend processes. As I have discussed prior, I was going to manipulate my already existing Axios get function for this. I stored my API axios request in a useEffect hook for this purpose, as the hook meant that the Axios fetch would run once when the page loaded, and again when any of its dependencies changed or updated. Thus, my plan was to create variables which housed the user's filtering options at any one point and place them in the hook's dependency array. This method meant that as soon as the user updated their filtering choices, the hook would run again and make another API request to read the database. If I didn't utilise useEffect, I would have had to manually trigger API requests for filtering updates, complicating the process with additional state management and event listeners.

Establishing Variables for User's Filtering Selections

However, I still needed to consider how I would get the filtering options to the database query, which was located in the Prisma code in the route.ts file of my api folder. To achieve this, I decided I would send the filtering options variables from the browse page to the api endpoint via URL. Due to the logic behind Next.js' API endpoints, the code inside the route.ts file itself only runs when the endpoint is called, meaning that it does not execute until there is an actual request from the frontend. Thus, I could include the variables for current filter selections in the URL when executing the Axios fetch and then use JavaScript's url.searchParams.get function to successfully retrieve the filtering options on the route.ts file. From there, I could again store these parameters in variables and use Prisma's language (which was conveniently very intuitive) to appropriately manipulate the database query. Now that I had a comprehensive plan for the backend, I could begin by creating the variables which would store the user's filter selections.

I started by adding a Boolean variable named onlyDiscount for the on sale filter, when the user clicked the switch, it would set onlyDiscount to the opposite of what it was so it was toggling it. I also included a checked={onlyDiscount} to ensure that onlyDiscount's value and the switch on the page remain in sync at all times.

```
<Switch onClick={() => (setOnlyDiscount(!onlyDiscount))} checked={onlyDiscount} />
```

Next, I needed to establish two variable which would keep track of the user's genre selections and price range selections, respectively. However, this was a little trickier as a user could make multiple selections at once rather than simply toggling a switch, and thus these variables would need to be an array.

For the genre array, I defined a variable named selectedGenres as an array of strings, which would track what genres have been selected by the user. I then created a function named handleGenreToggle which would update this array by either removing or adding a genre. When a genre is clicked, the function is called and checks if this genre is already in the selectedGenres array. If it is, the genre is removed using a filter, otherwise it is added to the array using the spread operator. This method results in an array which contains what genres the user has selected at any given time. In order to ensure that the checkboxes always match the selectedGenres array to avoid any confusion (eg. only action games showing, but action checkbox isn't checked), I added checked = {selectedGenres.includes(genre)}. I tested this code by logging the array after it updated on my local server and upon selecting a genre, it would be successfully added to the array and removed when clicked again. Thus, I copied this method for the price ranges and had all of the variables storing the user's filter selections complete. The only difference was that I assigned indexes to each of the price ranges (eg. 1: Free - $15, 2: $15 - $40 etc) so that the selectedRanges array was simplified rather than containing items like "$15 - $40".

```
const handleGenreToggle = (genre: string) => {
    setSelectedGenres((prevGenres) =>
        // Check if array of selected genres contains genre being clicked
        prevGenres.includes(genre)
            ? prevGenres.filter((g) => g !== genre)
            : [...prevGenres, genre]
    );
};
```

handleGenreToggle function

```
<Label
    htmlFor={`genre-${index}`}
    className="■text-offwhite font-interlight
    checked={selectedGenres.includes(genre)}
    onClick={() => handleGenreToggle(genre)}
>
    {genre}
</Label>
```

Code for genre filtering menu

Accomodating Axios Function for Filtering Variables

Next, I needed to add the filter selection variables to the Axios fetch URL. I updated the URL of the Axios fetch to from "/api" to "api?genres=${selectedGenres.join(',')}&priceranges=…". I also added each of the respective variables to the dependancy array so that the fetch would re-run each time a user changed their filter selections. For the genres and ranges arrays, I also needed to convert them to single comma seperated strings, which is necessary to properly format them as query parameters in the API URL.

```
// Retrieving information from api page, also sending specificities like filters etc
useEffect(() => {
    axios.get(`/api?genres=${selectedGenres.join(',')}&priceranges=${selectedRanges.join(',')}&discount=${onlyDiscount}`)
        .then(response => {
            const { games, totalResults } = response.data;
            setGames(games);
            setTotalResults(totalResults);
        })
        .catch(error => {
            console.error("There was an error fetching the game's information: ", error);
        });
}, [onlyDiscount, selectedGenres, selectedRanges]);
```

Querying Database According to Filtering Conditions

```
const url = new URL(req.url);
// Assigning variables to each of the values passed through api request on browse
const discountParam = url.searchParams.get('discount');
const discount = discountParam === 'true';
const genresParam = url.searchParams.get('genres') || '';
const genreArray = genresParam.split(',').filter(Boolean);
const priceRangesParam = url.searchParams.get('priceranges') || "1, 2, 3, 4, 5";
const priceRangesArray = priceRangesParam.split(',').map(Number);
```

Retrieving each filtering selection variables from the URL

Subsequently, I would need to accommodate my API endpoint which queried the database for this filtering. Firstly, I retrieved the variables from the URL using url.searchParams.get and stored them in respective variables. Prisma's extensive documentation[22] greatly assisted me during this time as it helped show how to customise the database query. I decided I would start with the discount only filtering since it seemed the most simple. In the database query, I added a 'where' parameter, and then checked discount's value from the URL, and if it was true then the query would only return database entries which had epic_on_sale = 1 or steam_on_sale = 1. Upon testing this by visiting '/api?discount=true', the API endpoint returned exclusively games on sale meaning it was successful.

```
try {
    const games = await prisma.games.findMany({
        where: {
            ...(discount && {
                OR: [
                    { steam_on_sale: "1" },
                    { epic_on_sale: "1" },
                ],
            }),
        }
```

Where clause for database query

Next, I needed to implement the querying for the price ranges. At this point, I decided I would create a variable in the Prisma file named whereClause and systematically add each filtering condition. This would make my code easier to read and maintain because it meant I could separate each of these sections out making the query less cluttered. For the price range querying section, I start by ensuring that whereClause exists or by initialising it as an empty array. Then, for each selected price range in priceRangesArray, I create conditions that filter games based on their steam_price or epic_price. Each price range index corresponds to a specific range (e.g., 1: Free - $15, 2: $15 - $40, etc). Using a switch statement, I define these conditions and use OR logic to ensure that a game falls within the specified range for either steam_price or epic_price. This logic means that a game, which has prices of $45 and $100 on Steam and Epic Games respectively, will be returned when the user selects a range which contains either of these prices. After mapping these conditions, I filter out any null values and add the resulting conditions to the AND array of whereClause, which ensures that the final query will only return games that meet the selected price ranges (if the user has selected ranges in filter menu).

```
const whereClause: any = {};
```

Defining whereClause object

```
try {
    const games = await prisma.games.findMany({
        where: whereClause,
```

Filtering by whereClause in database

```
// Apply price range filters if 'priceRangesArray' is not empty, each case corresponds to different price ranges,
// filtering based on 'steam_price' or 'epic_price'
if (priceRangesArray.length > 0) {
    whereClause.AND = whereClause.AND || [];
    const priceConditions = priceRangesArray.map((range) => {
        switch (range) {
            case 1: return { OR: [{ steam_price: { gte: 0, lte: 15 } }, { epic_price: { gte: 0, lte: 15 } }] };
            case 2: return { OR: [{ steam_price: { gte: 15, lte: 40 } }, { epic_price: { gte: 15, lte: 40 } }] };
            case 3: return { OR: [{ steam_price: { gte: 40, lte: 70 } }, { epic_price: { gte: 40, lte: 70 } }] };
            case 4: return { OR: [{ steam_price: { gte: 70, lte: 100 } }, { epic_price: { gte: 70, lte: 100 } }] };
            case 5: return { OR: [{ steam_price: { gte: 100 } }, { epic_price: { gte: 100 } }] };
            default: return null;
        }
    }).filter(Boolean); // Filter out null values
    whereClause.AND.push({ OR: priceConditions });
}
```

Adding price range filtering to whereClause

For the genres filtering, I start by checking again that the array is not empty, before adding it to the whereClause (as I did above). Then, I used whereClause.game_genre to specify a filter condition that involves the game_genre table, which is the linked table bridging games and genres in a many to many relationship (each game can have multiple genres). By using whereClause.game_genre, the filter targets this relationship, allowing the query to consider which genres are linked to each game. Using 'some' further refines this filter by specifying that a game should be included in the results if it has at least one associated entry in game_genre that satisfies the inner condition. The inner condition checks if any of the genres related to the game (in game_genre) have a genre record that

```
// If genreArray is not empty, then filters
// games by matching any genres in the 'genreArray'
if (genreArray.length > 0) {
    whereClause.game_genre = {
        some: {
            genres: {
                genre: {
                    in: genreArray,
                },
            },
        },
    };
}
```

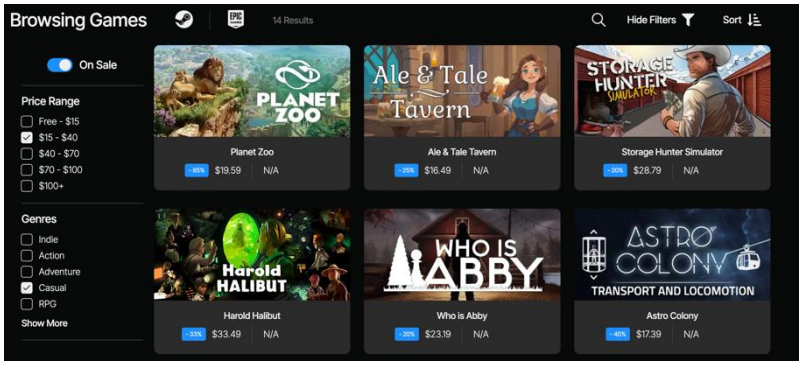
The inner condition I mention

Genres filtering conditions being added to

[22] https://www.prisma.io/docs/orm/prisma-client/queries/filtering-and-sorting

matches any value present in genreArray. This is achieved by searching the ‘genre’ field in the ‘genres’ table and using the keyword ‘in’ to check for a match between the genre name and the list of values in genreArray. Thus, the condition will return games which are associated with any of the genres in genreArray. I should also clarify that my filter system doesn’t require ‘exclusive’ matches with this logic (ie. if a user selects both action and adventure, then games with either of these will be returned, rather than exclusively games with both genres) because this approach caters to usability by offering a wider variety of games that cater to diverse interests.

I then did some testing by visiting ‘localhost:3000/api?...’ pages with various requests to ensure that the filtering system could appropiately handle any combination of filtering selections. All of my requests worked as expected and correctly functioned in tandem, returning the matching gmes for the conditions every time. Then, I went back into the browse page and played with filters to test that the games were being fetched correctly too, which was successful.



Games filtered by $15 - $40, discounted, and casual. Games displayed are correct

Implementing Search Functionality

Next, I needed to develop the backend for the searching functionality. There was two search bars: on the home page and a dynamic one on the browse page meaning I would also need to ensure that these were synced up and both functioned in tandem. I started with the search bar on the home page first, by creating a string variable named searchTerm. In the search bar code, I added these lines: value={searchTerm} and onChange={(e) => setSearchTerm(e.target.value)} meaning that whatever the user typed in the search bar would be stored as searchTerm at all times. In order to actually get this value from the home page to the browse page (and then to the API endpoint after that), I created a function named handleSearchSubmit which was called whenever the user submitted their search. This function prevents the default form submission behavior and instead navigates to the browse page with the search term as a query parameter in the URL, allowing the search term to be accessed on the browse page (which I do below).

```
<form className="w-full z-20 max-w-[380px] mx-auto"
/* Calling search function upon search submit */ onSubmit={handleSearchSubmit}>
    <div className="relative w-auto mx-5">
        <IoIosSearch className="absolute text-offwhite w-auto h-[70%] ml-3 top-1/2 trans
        <input
        className="w-full pl-19 p-4 text-[18px] placeholder:font-inter text-offwhite fon
        rounded-searchbox placeholder:text-grey ring-[1px] ring-lightermidnight placeh
        placeholder="Search for a game..."
        /* Setting the search term */
        value={searchTerm}
        onChange={(e) => setSearchTerm(e.target.value)}
        />
    </div>
</form>
```
Code section for search bar on home page

```
// When user submits a search, this function sends that variable via
// URL to browse page for search filtering
const handleSearchSubmit = (event: React.FormEvent<HTMLFormElement>) => {
  event.preventDefault();
  router.push(`/browse?search=${searchTerm}`);
};
```
handleSearchSubmit function

Subsequently, I retrieved the search term from the URL using the built in Next.js useSearchParams() function and defined the variable as searchTerm again. Next, I would need to manage the dynamic search bar on the browse page and ensure it was synced with this value. Thus, in this search bar, I set value={searchTerm} which meant that if a user searched ‘racing’, then the dynamic search bar would already have ‘racing’ in it. In order to change searchTerm’s value if the user typed in this dynamic search bar, I used the line onChange={(e) => setSearchTerm(e.target.value)}. This line updates the searchTerm state with the user's input whenever they type in the dynamic search bar, meaning that I only needed to worry about one variable for the search term rather than two separate variables. Finally, I added the searchTerm to the Axios fetch function by adding ‘&search={searchTerm’ and ensured to add the searchTerm variable to the dependancy array also. My final step for the search functionality was to actually accommodate the database query to filter for this value in the Prisma API endpoint.

```
const searchParams = useSearchParams();
const searchQuery = searchParams.get('search') || '';
const [searchTerm, setSearchTerm] = useState<string>(searchQuery);
```

Retrieving search term from home page and storing it as

```
<input
    type="text"
    value={searchTerm}
    placeholder="Search..."
    onChange={(e) => setSearchTerm(e.target.value)}
    className={`ease-in-out transition-all duration-200
    font-inter placeholder:text-grey focus:outline-nor
/>
```

Dynamic search bar code on browse page

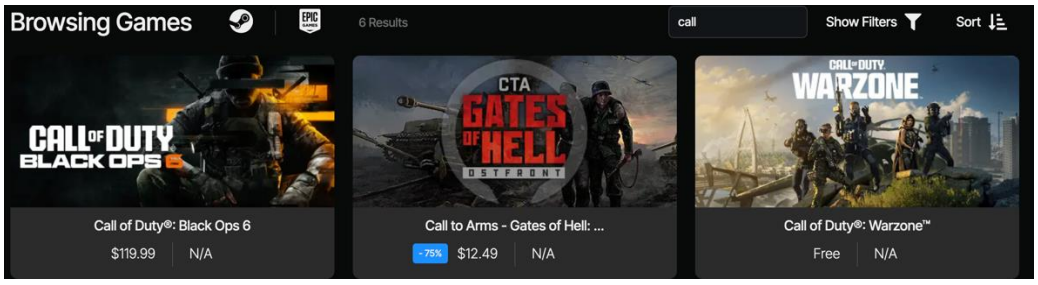
```
// Retrieving information from api page, also
useEffect(() => {
    setLoading(true);
    axios.get(`/api?&search=${searchTerm}&disc
        .then(response => {
            const { games, totalResults } = re
            setGames(games);
            setTotalResults(totalResults);
            setLoading(false);
        })
        .catch(error => {
            console.error("There was an error
            setLoading(false);
        });
}, [searchTerm, onlyDiscount, selectedGenres,
```

Updated Axios fetch sending searchTerm

In this Prisma file, I started by once again retrieving the search query with the line const searchQuery = url.searchParams.get('search');. I followed a similar process to the filtering conditions and added a new condition to the whereClause object. If a searchQuery exists, the code first makes sure that the AND condition array is set up (creating it if it doesn’t already exist). Following this, the condition specifies that only entries in the ‘games’ table (main table with game info) whose name field contains the searchQuery variable. I also ensure to set the query mode to ‘insensitive’ which means the string comparison between the searchQuery variable and the ‘name’ column will case-insensitive. For example, a search for "game" would match "Game," "game," "GAME," or "gAmE" in the name field, ensuring that users can find matches without needing to exactly match the stored values and improving usability hugely. Lastly, I quickly did some testing on my local server; both the home page and browse page search bars functioned correctly and returned appropiate results as shown below where I search for ‘call’.

```
// Filters games by checking if the 'name' contains
// the search query, case-insensitive
if (searchQuery) {
    whereClause.AND = whereClause.AND || [];
    whereClause.AND.push({
        name: {
            contains: searchQuery,
            mode: "insensitive"
        }
    });
}
```

Searching filtering conditions which are in whereClause



At this point, I also introduced a separate query in the Prisma file, which simply counted the number of games in the ‘games’ table which would be returned when whereClause was applied. Then, on my browse page, I displayed this value next to the Steam and Epic Games logos (as shown in image above) as it

```
// Count total results for 'x Results' on browse page
const totalResults = await prisma.games.count({
    where: whereClause,
});
```

Database query to count results

may be useful for users searching or filtering to know how many results are returned for their selections.

Implementing Sorting Functionality

My next task was to develop the sorting menu, since it's highly likely users may want to sort by things like price and popularity when trying to find video games. I started the same way I started the other filters; by adding a variable to track the user's sorting selection. I decided to ask some stakeholders which sorting options they would like on the site and track the answers to decide what options to add. I asked 6 stakeholders and all of them said both popularity and price (both low to high and high to low) sorting, four said alphabetical, one said by player count, and another one said by release date. After looking at these results, I decided I would have sorting options for popularity (default option), price (both ways again), and alphabetical. I ruled out player count because it was only mentioned by one stakeholder, and it would be difficult to track since it's constantly changing, making it less practical compared to more stable options like popularity and price. I also ruled out release date because it was only mentioned by one stakeholder, and it seemed less critical for users focused on finding popular and well-priced games.

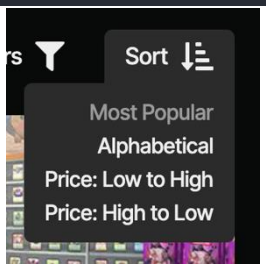
I started by creating an object named 'sorting' for the sorting options, so that each sorting option had its own id (eg. 1: Most Popular, 2: Alphabetical etc). I also created an 'orderBy' number variable to keep track of the id of the sorting option the user selected, and assigned it initially to one (most popular). In order to toggle the sort menu, I added a 'showSortMenu' boolean which was toggled whenever the user pressed the 'Sort' button. This boolean would conditionally render (if showSortMenu is true) a small box which would contain the sorting options. Subsequently, inside the box I mapped the 'sorting' dictionary and added {sort.label} so that each of the options would display. I also included an onClick={() => { setOrderBy(sort.id) }}, so that the 'orderBy' variable would constantly house the user's sorting selection. For visiblity of system status, I also made the sorting option text darker for the user's current selection with this check: '${orderBy == sort.id ?...'. Lastly, I added the 'orderBy' variable to the Axios fetch and its dependancy array to send it to the Prisma API endpoint on API requests.

```
{showSortMenu && (
    <div className="bg-lightmidnight mt-10 rounded-tl-md rounded-b-md right-0 flex flex-col absolute
    text-[16px] text-right font-inter px-4 py-2.5 gap-y-[1px] z-30">
        {/* Sorting menu and conditionally making them darker to show they are selected */}
        {sorting.map((sort) => (
            <p className={`underline-animation2 transition-all duration-150 cursor-pointer
            ${orderBy == sort.id ? 'text-darkerwhite' : 'text-offwhite hover:text-darkerwhite'}`}
            onClick={() => { setOrderBy(sort.id) }}>{sort.label}</p>
        ))}
    </div>
)}
```

Sorting menu code section

```
const sorting = [
    { id: 1, label: "Most Popular" },
    { id: 2, label: "Alphabetical" },
    { id: 3, label: "Price: Low to High" },
    { id: 4, label: "Price: High to Low" },
];
```

'sorting' object



Sorting menu on page with 'Most Popular'

On my Prisma API endpoint, I started by retrieving the user's current sorting selection from the URL and storing it as an integer variable named 'order'. Next, I initialised a new object named 'currentOrder'; I wasn't using whereClause here as in the database query, I would need to include this sorting condition in the 'orderBy' function instead of the usual 'where' function. Despite this, I was still going to use an object to store the condition so that I could keep the database query code itself as clean and concicse as possible for future proofing. Next, I simply used if statements to check the current value of 'order' and assigned currentOrder to the appropiate sorting option. For example, if order = 2 (alphabetical id), currentOrder will be assigned to { name: 'asc' }. Lastly, in the database query, I added orderBy: currentOrder so that the query would return results in the appropiate order selected. To ensure this system was operational, I did some testing on the browse page and all four sorting options functioned correctly and still worked as expected in tandem with filtering (eg. selecting order by price low to high then filtering by a

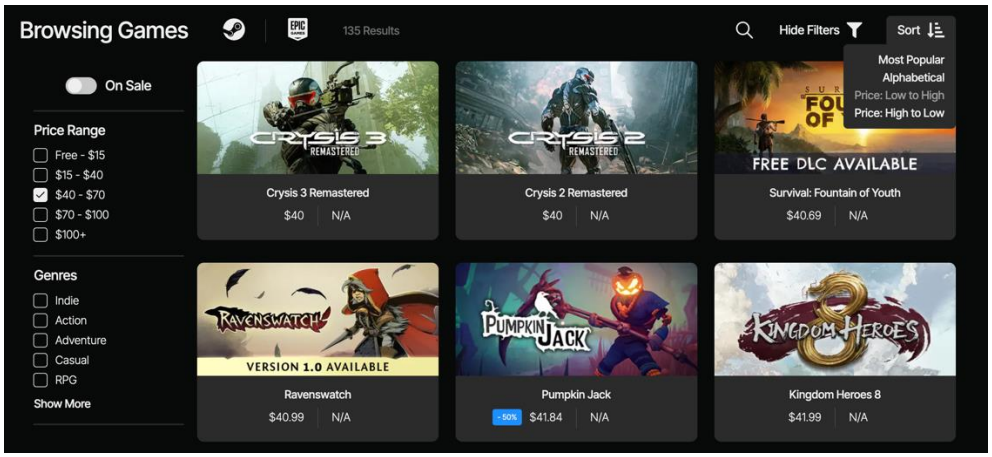
```
let currentOrder: any = {};
// Set currentOrder based on the value of 'order'
if (order === 1) {
    currentOrder = { game_id: 'asc' };
} else if (order === 2) {
    currentOrder = { name: 'asc' };
} else if (order === 3) {
    currentOrder = { steam_price: 'asc' };
} else if (order === 4) {
    currentOrder = { steam_price: 'desc' };
}
```

Assigning currentOrder based off user selection

```
try {
    const games = await prisma.games.findMany({
        where: whereClause,
        orderBy: currentOrder,
    });
```

Ensuring database query orders results by currentOrder

price range would display the games in the price range from lowest to highest as intended). An example of this working is also shown below.



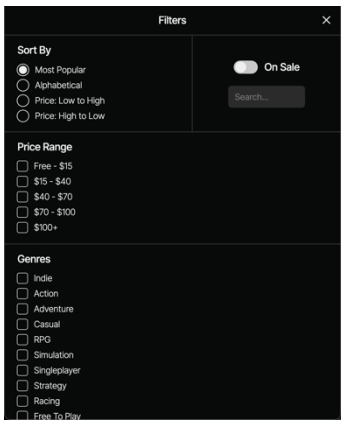
Applying $40 - $70 filter and sorting by lowest price shows $40 games first as expected

Developing the Responsive Filtering Menu

Now that my filtering system was complete, I needed to create a filtering menu and the hamburger menu on smaller screens to better adhere to accessibility. I started with the filtering menu simply because I had just finished the regular screen filtering so I would know exactly how my code around this functionality worked. After looking around some other websites like Nike, Adidas, and Stockx, I found that the industry standard approach to responsive filtering menus was a full screen menu which contained all filtering, sorting, and searching functionality. Despite this, I still wasn't sure about this design choice as it meant that users would be unable to see any of the main content whilst filtering on small screens. Another potential option was a similar menu which covered only ~60% of the screen similar to PB Tech's design, however this menu would be limited in terms of space which could compromise aesthetics and potentially mean it would need to be scrollable.

I decided I would leave the decision up to stakeholders as I believed both designs had their pros and cons. I ended up asking six stakeholders whether they preferred the full screen menu or the ~60% menu by getting them to play around on both Nike's and PB Tech's websites respectively. However, after all six stakeholders agreed that Nike's design was superior, I realised that this survey was likely prone to bias as Nike's website was, in general, superior to that of PB Tech in terms of general design. This likely led to stakeholders basing their decision off Nike's superior aesthetics rather than actually preferring the full screen menu to the 60% menu making my survey ineffective for gathering accurate results. For this reason, I decided I would create both versions on my own website and repeat the survey with these.

To create this responsive filter menu, I started by adding a respMenu boolean which was set to true whenever the user clicked the 'Filter' button. To make the menu have a sliding up animation, I made the filter menu have the 'top-[100vh]' property so it was essentially constantly just below the user's current viewing point, and added 'translate-y-[-100%]' when respMenu was set to true to make it pop up with the appropiate animation with Tailwind CSS. I added the X in the top right which set respMenu to false, and then began working on the filters themselves. Next, for each of the filtering categories, I simply copied over my code from the regular filter menu and search bar. For the sorting menu however, I switched to a radio menu (similar to checkboxes, but only allow one choice at a time, added via shadcn/ui component) because the text alone (like in the regular



Responsive filtering menu

sorting menu) looked out of place when around other checkboxes, and wasn't immediately clear what it was. This switch was fairly simple however as I could map the same object 'sorting' for the radio menu code. My next job was to 'sync' all of the filters; for example if a user checked the 'action' genre and '$40 - $70' price range checkboxes, then these checkboxes should still be checked on the responsive filter menu too. This should be the case for every single thing on the responsive filter menu because it ensures a consistent user experience, allowing selections to persist across different views without needing to reapply them.

Syncing Filter Options Across Menus

Firstly, I ensured that the current sorting selection would be synced by using value={orderBy.toString()} (shadcn's value had to be a string value), meaning that if a user selected alphabetical (id 2) on the regular sorting menu, then the second radio menu field would sync and pre-selected already. From there, to ensure that the radio menu could also be used to change the sorting selection, I added onValueChange={(value) => setOrderBy(parseInt(value)), so that whenever the user changed the radio menu's selection, orderBy's value would update with it. Additionally, since I was using the same variable as before, I didn't need to change the backend at all. Upon testing between the regular and responsive sorting menus, both menus were synced on the frontend due to this dynamic approach of checking orderBy's value to determine which sorting option was active.

```
{/* Responsive radio menu for sorting */}
<RadioGroup value={orderBy.toString()} onValueChange={(value) => setOrderBy(parseInt(value))}
    {sorting.map((sort) => (
        <div key={sort.id} className="flex items-center">
            <RadioGroupItem value={sort.id.toString()} id={sort.id.toString()} className="h-5
            <Label htmlFor={sort.id.toString()} className="text-offwhite font-interlight ml-
                {sort.label}
            </Label>
        </div>
    ))}
</RadioGroup>
```

Code section for sorting radio menu (responsive filter menu)

Secondly, I needed to sync the genres and price ranges (did these at the same time due to their similarity) with the regular filter menu. In order to achieve this, I would need to define each checkboxe's status (active vs inactive) not based off whether they have been clicked, but rather whether or not their respective value is in the 'selectedGenres/selectedRanges' array. Thus, I added the line 'checked={selectedGenres.includes(genre)}' (same thing for ranges but different variable and array names) which simply checked for this condition and if it was true, ensured the checkbox was checked. I needed to add this to both the responsive and regular filter menu sections so that both were always synced, and upon testing this addition solved the problem, ensuring users could always be aware of what filters they selected even when changing screen sizes.

```
{genres.map((genre) => (
    <div key={genre} className="flex items-center mr-5">
        <Checkbox id={genre} checked={selectedGenres.includes(genre)}
        onClick={() => handleGenreToggle(genre)} />
        <Label htmlFor={genre} className="text-offwhite font-interli
            {genre}
        </Label>
    </div>
))}
```

Code section for the genres checkboxes on the responsive filter menu. The ranges section is identical besides the variable names

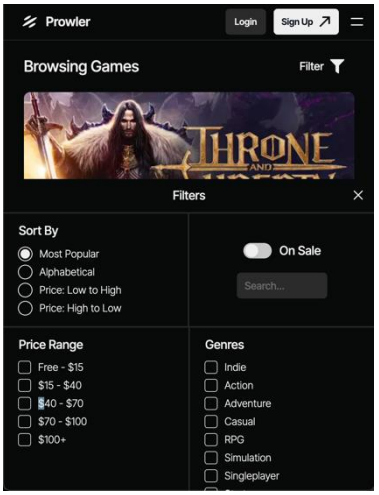
Lastly, I needed to sync both the search bar and the 'on sale' switch. For the 'on sale' switch, I added 'checked={onlyDiscount}' to both menus, meaning the switch would be active if the variable was true, or inactive otherwise and this was all that was needed. For the search bar, I didn't end up needing to change anything because it already contained 'value={searchTerm' due to the syncing I did between the home page and browse page search bars. Again, I didn't need to change anything else since the variables I was using were the same. Upon testing, both of these functions were synced up, meaning I moved onto making a ~60% height version.

```
{/* Responsive on sale filter and dynamic search */}
<div className="flex items-center">
<Switch onClick={() => (setOnlyDiscount(!onlyDiscount))} checked={onlyDiscount} />
    <p className="ml-3 text-[19px] text-offwhite font-inter">On Sale</p>
</div>
```

Code section for 'on sale' switch on responsive filter menu

Deciding on Filter Menu Design

This was reasonably quick as all I needed to edit was the layout of the menu; I moved the genres section to be in line with the price ranges. As I expected, I had to make the menu scrollable as the genres exceeded the menu's height in this version, however you were able to see the main content behind it. I proceeded to show the same six stakeholders both versions and allow them to use the menus as they pleased, and asked them which menu they preferred. This time, two people said they preferred the ~60% version and four said the full screen version, proving my hypothesis correct and providing valuable insight for this decision. Overall, I chose to stick with the full screen menu design because it aligned with industry standard and offered the necessary space for all filtering options without compromising the visual appeal or the need to scroll. On top of this, users not being able to see the main content while filtering isn't crucial for my site, as they are typically focused on refining their search results anyway.



Implementing Responsive Hamburger Menu in Navbar

Back in the nav.tsx component file, I decided I would create an object for the navbar links as I already had three similar lines for the three navbar links, and I was about to add a similar menu. Thus, I created a navLinks object where I assigned each page's label with its link (eg. browse : ../browse). For the links on bigger screens, I changed the code from three separate lines to one line, mapped the navLinks object, and changed the text and link to dynamic like item.link which functioned identical to before, but now was more concise and future-proofed.

```
const navLinks = [
  { label: "Browse", link: "../browse" },
  { label: "Contact", link: "../contact" },
  { label: "GitHub", link: "https://github.com/kba0297/Prowler" },
];
```

Defining navLinks object

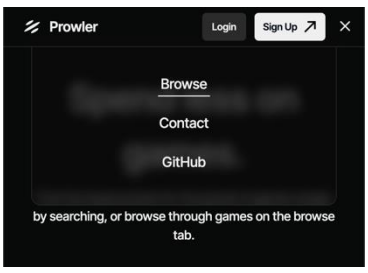
```
{navLinks.map((item) => (
  <a className="font-intersemibold text-[15px] text-grey hover:text
  duration-200 underline-animation1" href={item.link}>{item.label}</a>
))}
```

Mapping object to dynamically render the 3 links in

Next, for the hamburger menu I created a boolean named hamburgerMenu which was toggled whenever the user pressed on its icon in the top right. I decided on a hamburger icon just because it followed industry standards and so users could immediately recognise what the button's purpose was. I was going to deploy a simple small dropdown menu for this menu the login functionality buttons would remain visible on small screens for easy access (and because they are smaller so can still fit there on thin screen), thus this menu would only contain the navbar links. I also decided to make the menu's background slightly transparent as I did with the navbar so that the whole thing kept a consistent look to it. Then, in the menu, I copied the mapped code for the navbar links and changed the styling so that they were in rows rather than in columns to save width. Lastly, I added a small underline effect to both sets of navbar links when

```
// Responsive navigation menu animations (presets in tailwind.config file)
${hamburgerMenu ? 'animate-menudown pointer-events-auto' : 'animate-menuup pointer-events-none'}
${pressedMenu ? 'block' : 'hidden'}`}>
  <div className="flex mt-4 flex-col w-full absolute">
    {/* Mapping the different links in responsive navigation menu to make code more concise */}
    {navLinks.map((item) => (
      <a className="flex items-center self-center px-4 mt-4.5 underline-animation"
      href={item.link} onMouseOut={handleMouseOut}>
        <p className={`font-inter text-offwhite text-[20px] px-2 py-1 z-20 transition-opacity
        ${hamburgerMenu ? 'opacity-100' : 'opacity-0'}`}>
          {item.label}
        </p>
      </a>
    ))}
  </div>
</div>
```

Code section for responsive hamburger menu



Hamburger menu when active

hovered for visibility of system status and for a more interactive feel to the site. I also used Tailwind CSS to change the hamburger menu icon to an 'X' indicating that users could click it again to close the menu if they pleased, for which I got the idea from linear[23], which also addresses help and documentation better.

Game Page Development

Now that I had mostly finished my browse and home pages, it was time to develop the final dynamic page (contact and FAQs would be static), the games page. This page was for when users clicked on a game on the browse page to find out more information about it. Because I was only going to be querying the database for information about one particular game, I decided I would create a separate database query (aka a separate API endpoint) which only retrieved information about the game to minimise resources used, thus lowering load times.

I started by creating a new page and followed the identical skeleton for my other pages which was mostly boilerplate code. When developing this page, I decided that I was going to set up the database query and retrieval of game information before designing the content of the page to simplify the process, as I could dynamically code at the same time. Thus, I setup a 'games' folder in my 'api' folder and created a route.ts endpoint inside. Again, I copied and pasted most of the boilerplate code (like error handling, imports) from the browse page's Prisma file to use for this one. This games page wasn't going to have any variables which would change the query and thus all I needed to do in this endpoint was to return all database info for the id of the specified game, meaning all I needed to do was fetch the game's id which had been clicked.

Retrieving Game ID and Fetching its Details

In order to retrieve this id to use on the game's page database query, I was going to follow a similar method to my search query; as in I would send the id from the browse page to the game page via URL, and finally to the API endpoint via Axios fetch. Firstly, back in the browse page, I added 'href={'/game?id=${game.game_id}'}' to the parent div of each game's card, meaning whenever a user clicked a game card, they would be sent to that URL. Subsequently, on the game page file, I used searchParams to retrieve the id and stored it as a variable named 'id'. Next, I created my Axios fetch on the game page, including 'id=${id}' in the URL to send this value to the API endpoint, meaning the link would be '/api/games' rather than simply '/api'. I didn't need to include the id variable in the dependency array for the useEffect function since the game id would never be updating when the user was on this page.

```
// When a game is clicked, pass the game id to take user
<a href={`/game?id=${game.game_id}`} key={game.game_id}>
```

Parent div of each game card

```
// Axios statement to retrieve data from games api page (in /api/games)
// for the specific game's id, with loading logic too
useEffect(() => {
    const fetchData = async () => {
        try {
            const response = await axios.get(`/api/games?id=${id}`);
            const selectedGame = response.data;

            if (selectedGame) {
                setGame(selectedGame);
            } else {
                console.error("Game not found");
            }
        } catch (error) {
            console.error("Error fetching game:", error);
        }
    };

    if (id) {
        fetchData();
    }
}, [id]);
```

Code section for Axios fetch on game

[23] https://linear.app/

Now on my API endpoint/Prisma file for the game page, I again used searchParams to retrieve and store the game's id from the URL as a variable named id. Next, in the database query, which was a constant named 'game', I changed the query from games.findAll to games.findUnique so that the query could only return one result. In the where condition for the query, I added game_id: parseInt(id, 10) } (id is stored as string when retrieved from URL), meaning that the query would look for and only return the entry in the 'games' table which had an id that matched the id variable, and thus the game which the user clicked on (and all of its details) would be returned by the query. To also fetch the genres associated with the game, I utilized the include property in the Prisma query. By specifying include: { game_genre: { include: { genres: true } } }, the query not only retrieves the game details but also includes the relevant genres ids from the game_genre table. I then created a formattedGame object that includes all the original game data and maps over the game_genre array to extract the genre names, which are then included in the JSON as the 'genres' property. This results in a much cleaner structure where the genres property contains an array of genre names, making it easier to work with on the game page (structure for genres property wihout the formattedGames vs with shown to the right). Lastly, I returned the formattedGame object as JSON at the API endpoint for use on the game page.

```
try {
    // Query the database to find a game by its unique 'game_id'.
    // Also include the related 'game_genre' and 'genres' in the response.
    const game = await prisma.games.findUnique({
        where: { game_id: parseInt(id, 10) }, // Parse 'id' to an integer for the query
        include: {
            game_genre: {
                include: {
                    genres: true, // Include the genres related to the game.
                },
            },
        },
    });
```

Database query for specific game id

```
// Format the 'game' object to include a simplified 'genres' array.
// Map over 'game_genre' to extract the 'genre' names.
const formattedGame = {
    ...game,
    genres: game.game_genre.map(({ genres }) => genres.genre),
};
```

formattedGame object which adds 'genres' property

```
"game_genre": [
  {
    "id": 1,
    "game_id": 2,
    "genre_id": 2,
    "genres": {
      "genre_id": 2,
      "genre": "Action"
    }
  }
]
```

Without formattedGames

```
"genres": [
  "Action"
]
```

With

Developing Game Page Frontend

Following this, I returned to the game page to begin developing the frontend. During this phase, I was mostly copying my design on Adobe XD as this page was fairly simple. Whenever I added dynamic game information, since I had already setup the query and stored the information from the API endpoint in a object named 'game', I simply used {game.x} where x is the name of a column in the database (this is for most fields, some require formatting which I describe below). Since I had scraped many images for each game, and then stored them in an array, I decided I was going to implement a carousel on the game page for these images. As I did on the home page, I used shadcn/ui's carousel component for this and added the loop:true property to the carousel, meaning it would loop back to the first item after the last item. Inside of the <CarouselContent> tag, I mapped the images array, ensuring to also include the index in the mapping function. Inside of this .map, I added a <CarouselItem> tag and for each item, an image with src={url} meaning that each image from the game's steam page being scraped will appear in the carousel. I also ensured to add an alt tag to the image: 'Game Screenshot {index + 1}' for accessibility purposes, so any visually impaired users can navigate my site properly.

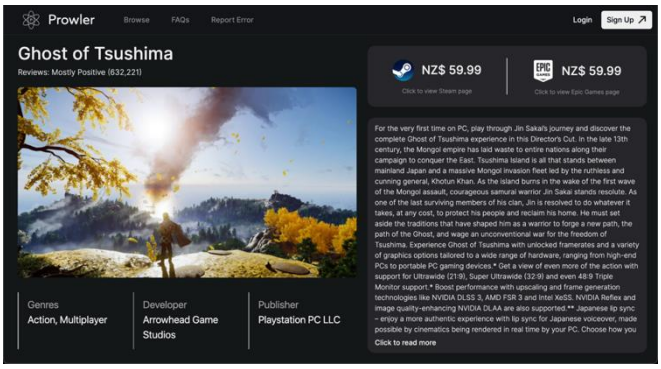
```
{/* Mapping game images to carousel items to infinitely loop through them */}
{imageUrls.map((url, index) => (
    <CarouselItem key={index}>
        <img src={url} alt={`Game screenshot ${index + 1}`} className="rounded
    </CarouselItem>
))}
```

Inside of the <CarouselContent> tag code

Handling Price Display and Description Length

A potential issue I noticed when adding the platform's respective prices was that often times, Epic Games did not sell the game and thus the epic_price value would be -1. Also, the reason for this is because my price fields are floats, which is the case because, in some cases, I need to do calculations with the price values (eg. calculating discount %) so storing these values as floats simply made things easier in the long run. However, I didn't want games prices displaying -1, so I added a check for if epic_price = -1, then the text for the price should be N/A, otherwise price value. I also did this for free games, since before they

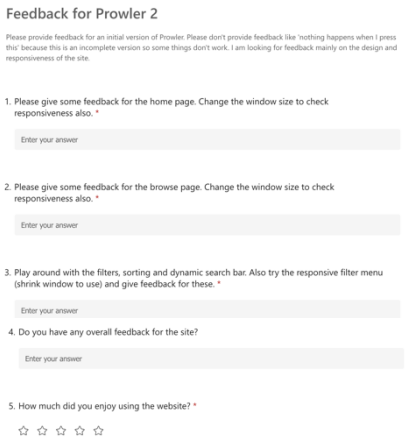
were displaying prices of 0, so I just checked both steam and epic price values, and if they were 0, then the text would render as ‘Free’, otherwise the price value (copied this check for browse page prices also). Lastly, I also clamped the long description’s length to 20 lines in order to prevent the description from going longer than the page and looking odd. I also added a button to read more so that users could read the whole description if they wanted to so that this clamping didn’t compromise the site’s functionality. I added this simply by checking if the description had excedeed 20 lines, and if so then add this button which toggled the clamping limit. The first finished iteration of the games page is shown to the right.



Second Widescale Stakeholder Feedback

At this point, it was Term 3 Week 4 meaning it was time for my second round of stakeholder feedback. Once again, I would be utilising a live testing server and microsoft form for this process to make it more efficient. I started by deploying another live version of my website using Vercel (no longer live) using the ‘vercel deploy’ command. This time, instead of needing to fix many errors, it deployed on my first try due to my increased knowledge around what was causing deployment errors. For example, previously I had received a ‘Parameter ref implicitly has an any type’ error when deploying which happens because TypeScript requires all parameters to have explicit types. Here, the ref parameter is missing a type, so TypeScript defaults to any, causing the error. Since ref references a DOM element, it should be typed as React.RefObject<HTMLElement>, ensuring TypeScript understands its purpose and provides proper type checking, which I now understood.

Next, I created a Microsoft form[24]. This time, I ensured that each question was shorter whilst still asking for feedback on specific parts of my website. This time, I asked users to play around with the filtering, sorting, and dynamic search bar and give feedback for these aspects of my site as I had just finished developing them meaning user testing was vital. I also specifically asked users to shrink their browser to test responsiveness since I viewed this as extremely important for my site and feedback would be beneficial. Lastly, I also moved the website link from the form to the email as last time someone was unable to find the website link since it was in the description of the form to make the process as easy as possible for my stakeholders. I ended up sending this form out to twelve stakeholders and received ten submissions. This round of feedback was generally even more positive than the last, and I received various compliments on the website’s aesthetics, usability, and user considerations. I did still however receive my fair share of feedback which pointed out various work-ons and potential improvements for the website.



Feedback: “The search bar on the browse page barely works, when typing the page is updating the results too much and it glitches out”

Implementing a Debounced Search

Upon typing one or two letters, the browse page would update and show the expected games. However, when typing quickly, the search bar would send an API request for each letter typed in rapid succession which resulted in many requests being sent in a short amount of time. This caused searches with many characters to take long times to load, and

[24] https://forms.office.com/r/VKCPNQAVCq

sometimes glitch out (shown in screenshot) and not load properly which was a huge issue. I researched this problem and found it was fairly common for on key press search bars and a common approach to solve this is to debounce the search input, which will limit how often the search function is called. This way, the API request only triggers after the user stops typing for a short duration.

```
GET /api?pageNumber=1&search=c&discount=false&genres=&priceranges= 200 in 1486ms
GET /api?pageNumber=1&search=cal&discount=false&genres=&priceranges= 200 in 2633ms
GET /api?pageNumber=1&search=call%20&discount=false&genres=&priceranges= 200 in 2356ms
GET /api?pageNumber=1&search=call%20o&discount=false&genres=&priceranges= 200 in 2201ms
GET /api?pageNumber=1&search=ca&discount=false&genres=&priceranges= 200 in 2738ms
GET /api?pageNumber=1&search=call&discount=false&genres=&priceranges= 200 in 2525ms
GET /api?pageNumber=1&search=call%20of&discount=false&genres=&priceranges= 200 in 2541ms
GET /api?pageNumber=1&search=call%20of%20duty&discount=false&genres=&priceranges= 200 in 2313ms
GET /api?pageNumber=1&search=call%20of%20du&discount=false&genres=&priceranges= 200 in 2317ms
GET /api?pageNumber=1&search=call%20of%20d&discount=false&genres=&priceranges= 200 in 2317ms
GET /api?pageNumber=1&search=call%20of%20dut&discount=false&genres=&priceranges= 200 in 2317ms
GET /api?pageNumber=1&search=call%20of%20&discount=false&genres=&priceranges= 200 in 2319ms
```

Screenshot of my terminal when searching for ‘call of duty’ showing the excessive API requests as search updates. Final request glitched; should contain ‘duty’ but doesn’t

In order to debounce the search term on the dynamic search bar, I was going to use the ‘lodash’ library as it includes a debounce function. Thereafter, I created a new function named updateDebouncedSearch using React’s useCallback function, which ensures that the function reference remains the same between renders. Inside the function, it wraps the setDebouncedSearchTerm function in a debounce with a 400ms delay, so that updates to debouncedSearchTerm only occur after the user stops typing for 400 milliseconds, reducing the frequency of updates. Subsequently, I added a useEffect function which called the updateDebouncedSearch function and parsed the searchTerm variable (so in the function, ‘term’ is just the searchTerm), and added searchTerm to its dependancy array. Finally, I updated the Axios fetch by switching out searchTerm for debouncedSearchTerm in the URL and dependancy array so that an API request would only be sent after the user stopped typing for 400ms. After testing, this fixed the issue perfectly and the search functionality was complete.

```
const [debouncedSearchTerm, setDebouncedSearchTerm] = useState(searchTerm);
```

Defining debouncedSearchTerm to be searchTerm initially

```
// Debounced function to update the debouncedSearchTerm state
const updateDebouncedSearch = useCallback(
    debounce((term) => {
        setDebouncedSearchTerm(term);
    }, 400),
    []
);

useEffect(() => {
    updateDebouncedSearch(searchTerm);
}, [searchTerm]);
```

Feedback: “I don’t like the long pauses between selecting a filter and the games changing, it makes it feel like when you choose something the page isn’t doing anything”

Adding Loading Status

I was unable to directly shorten the loading time when changing filters or sorting as this extended loading time was from Prisma’s database query. This complaint is fairly common on many websites which query databases, and the industry standard fix is to implement some form of loading indicator. This fix addresses visibility of system status as it lets users know that the page is updating and not just broken. I also researched the issue and found an interesting study[25] which proved through comprehensive testing that participants perceived services with loading indicators to be 11% faster.

```
{loading ? (
    <div className="mx-auto flex flex-col items-center text-offwhite text-[18p
        <p className="text-center mb-4">Fetching games..</p>
        <ScaleLoader color="#EFEFEF" height={20} margin={2} width={3} loading/>
    </div>
) : (
```

Conditionally rendering loading content

```
// Retrieving information from api page, also
useEffect(() => {
    setLoading(true);
    axios.get(`/api?pageNumber=${pageNumber}&se
        .then(response => {
            const { games, totalResults } = re
            setGames(games);
            setTotalResults(totalResults);
            setLoading(false);
        })
        .catch(error => {
            console.error("There was an error
            setLoading(false);
        });
}, [pageNumber, debouncedSearchTerm, onlyDiscou
```

Axios fetch for browse page where I change loading’s value based off response status

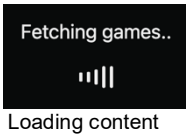
[25] https://www.chrisharrison.net/projects/progressbars2/ProgressBarsHarrison.pdf

In order to implement a loading indicator, I simply added a boolean named loading which was set to true at the start of each useEffect function where I was calling the API endpoint with Axios. Then, upon successfully retrieving the data, loading would be set to false and I added this logic to both dynamic pages (the browse and game page). Then, in order to display the loader, I imported 'ScaleLoader' from a react spinners website[26], similar to how I was importing icons from react icons. Lastly, on the browse and game page, I would conditionally render this loader if loading was true. The final result was that as soon as a user updated anything in the dependancy array (ie filters, sorting, search), the loader would appear along a small message like 'Fetching Games' to let users know that their change did update the results, and make the site seem faster.

Feedback: "If I scroll down on the browse page, I can't use the filters, sorting, or search bar which is a bit irritating needing to scroll back up"

Implementing Pagination

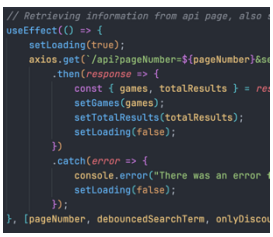
Currently, my browse page was utilising an infinite scroll to display games, meaning when users scrolled a far distance, they would have to scroll very far up to use any functionality. To fix this, I decided I would switch to a pagination approach. When considering how to implement pagination, it was possible to retrieve all database entries from the API endpoint, and then use JavaScript on the browse page to only display x number of games per page. However, I realised that if I was only going to be using x entries on the page, retrieving every single entry (which would be in the thousands) was likely highly inefficient when I could instead only query for x entries.



Loading content

Pagination Backend Logic

I started by creating a variable named pageNumber on the browse page which would track the current page value the user was on, and then added this number to the Axios fetch and the dependancy array. Now on the Prisma API file for the browse page, I once again retrieved this variable from the URL using searchParams. Next, I added a variable named itemsPerPage, which would determine how many games would be on each page, which I set to 48 for now. In the database query, I added 'take: itemsPerPage', which simply limited the query to only returning the first 48 entries from the database. Lastly, I added 'skip: (pageNumber – 1) * itemsPerPage', which meant that the query would skip over entries from previous pages, ensuring that each page displayed the correct set of games based on the current page number. Upon testing by visiting '/api?pageNumber=3', the endpoint returned 48 games, starting from id 97 meaning this logic was working correctly.



Axios fetch with pageNumber

Prisma database query for browse page

Planned design for pagination menu

[26] https://www.davidhu.io/react-spinners/storybook/?path=/docs/scaleloader--docs

My initial plan for the pagination menu was to have the current page number, plus and minus one number (ie if page number is 2, then have 1 and 3 also), and previous and next buttons. Because these values for page numbers in the menu were going to be dynamic, but also needed to follow some rules (eg. cannot be <= 0, or greater than max page number), I created an array of these clickable page number values in the menu named paginationItems. To populate this array, I added some conditions: if pageNumber is > 1, add (pageNumber – 1); add pageNumber; and if pageNumber < totalPages, add (pageNumber + 1). I calculated totalPages simply by ceiling dividing the total results value by 48. After this, I sorted paginationItems by ascending values, and then filtered out any negative or 0 values (these should never appear, but just in case). Thus, the paginationItems array would contain pageNumber and pageNumber plus/minus one. For example, for pageNumber = 3 then paginationItems = [2,3,4] but for pageNumber = 1 then paginationItems = [1,2]. In this section of code, I also define some conditions for later: isFirstPage which is true when pageNumber = 1, and isLastPage which is true when pageNumber = total pages.

```
// Pagination backend
const isFirstPage = pageNumber === 1;
const isLastPage = Math.ceil(totalResults / 48) === pageNumber;

let paginationItems = [];

const totalPages = Math.ceil(totalResults / 48);

if (pageNumber > 1) {
    paginationItems.push(pageNumber - 1);
}

paginationItems.push(pageNumber);

if (pageNumber < totalPages) {
    paginationItems.push(pageNumber + 1);
}

paginationItems.sort((a, b) => a - b);

paginationItems = paginationItems.filter(page => page > 0 && page <= totalPages);
```

Pagination Menu

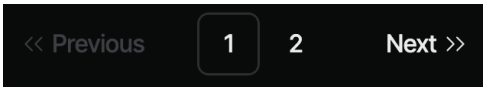
Next I needed to develop the pagination menu itself. The next and previous buttons were fairly simple to add since they were simply buttons which changed pageNumber's value by +1 or -1 respectively. I also checked isFirstPage's value (defined above) for the previous button, and if it was true then I greyed out the button and made it unclickable since there were no previous pages (and did the same but with isLastPage for next button). For the numbered buttons, I started by mapping the paginationItems array (passing 'page' as current item) and added a button inside this mapping function which housed each page number from paginationItems. I also checked if 'page' was equal to pageNumber (the true page number), and if so then I applied a lighter border for visibility of system status so that users always know what page they are on. I added an onClick to the button which would set pageNumber to 'page' meaning if users clicked page 3 in the menu, pageNumber would then be changed to 3. Finally, I added some arrow icons to the previous and next buttons for recognition rather than recall and because it was an industry standard practice.

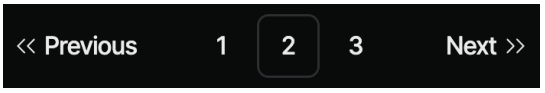
```
<a href="#" className={`flex mr-3.5 items-center font-i
${isFirstPage ? '☐text-[#4f4f54] pointer-events-none'
onClick={() => goToPage(pageNumber - 1)} >
    <IoIosArrowBack/>
    <p className="ml-1 mr-2">Previous</p>
</a>
```

Code for previous button (next button is near

```
{/* Mapping pagination item array for the page numbers at the bottom to display current page and +- 1 */}
{paginationItems.map((page, index) => (
    <a key={index} href="#" className={`☐text-offwhite flex mr-1.5 items-center ☐hover:bg-lightmidnight
    transition-colors duration-200 text-[16px] justify-center h-[38px] w-[38px] rounded-md
    ${page === pageNumber ? 'outline outline-[1.5px] ☐outline-lightermidnight' : ''}`}
    onClick={() => setPageNumber(page)}>
        <p>{page}</p>
    </a>
```

Code for the clickable page numbers





Pagination menu on page 1 vs page 2

At this point, I wanted to receive some more stakeholder feedback (on a smaller scale) for this pagination. I gave my laptop to three stakeholders, who agreed it was implemented well, however there were two possible improvements mentioned. Upon visiting later pages and then applying a filter, the page wouldn't return any games at all. The second improvement was that the previous and next buttons were somewhat pointless because you could simply use the numbered page buttons, and it was also hard to quickly get to pages far away (can't quickly get to page 10 from page 1).

Firstly, the root of the first problem was that as the user applied filters, the number of total results will shrink. This means that if a user was on page 30 before, and applies a filter, and the results shrink so that there are only 25 pages, then page 30 will have no results.

```
<Checkbox id={range.label} checked={selectedRanges.includes(range.id)}
onClick={() => (handleRangeToggle(range.id), setPageNumber(1))}/>
```

The most obvious and industry standard fix to this issue was to reset the page number to 1 whenever the user changed filters. My first instinct was to create a useEffect function which would reset pageNumber to one, and include all of the filtering/sorting/searching variables in the dependancy array which I tried. However, upon testing, I noticed that this was causing many issues and not working as intended. This is because, say the user updated variable selectedGenres, then both the Axios fetch and the setPageNumber(1) functions would be called synchronously. However, because pageNumber is also a dependancy for the Axios fetch, then the fetch function would be called twice in rapid succession (since pageNumber would update right after selectedGenres did and both are dependencies), which often causes errors with the API request. Thus, I simply went through the browse file and manually updated pageNumber to 1 each time something which affected the API request changed. For the same reason, as to minimise multiple API requests in rapid succession, I also made the filtering and sorting sections have no pointer events when loading was true, meaning that users couldn't change filters until their first change had finished loading (this wouldn't be frustrating because the loading times are very short).

```
<div className={`${loading ? 'pointer-events-none' : 'pointer-events-auto'}`}>
```

Improving Pagination Menu

Secondly, I agreed with the feedback about the previous and next buttons so I decided to change these to first and last buttons instead. This was a quick fix as I could just switch these buttons from adding/removing one from the pageNumber to setting the pageNumber to the first and last pages, respectively. For the last page, I again used the ceiling division (rounding up) of total results/48, and this change was done and working.

```
<a href="#" className={`flex ml-3.5 items-center font-inter trans
${isLastPage ? 'text-[#4f4f54] pointer-events-none' : 'text-c
onClick={() => setPageNumber(Math.ceil(totalResults / 48))}>
    <p className="mr-1 ml-1.5">Last</p>
    <RxDoubleArrowRight />
</a>
```

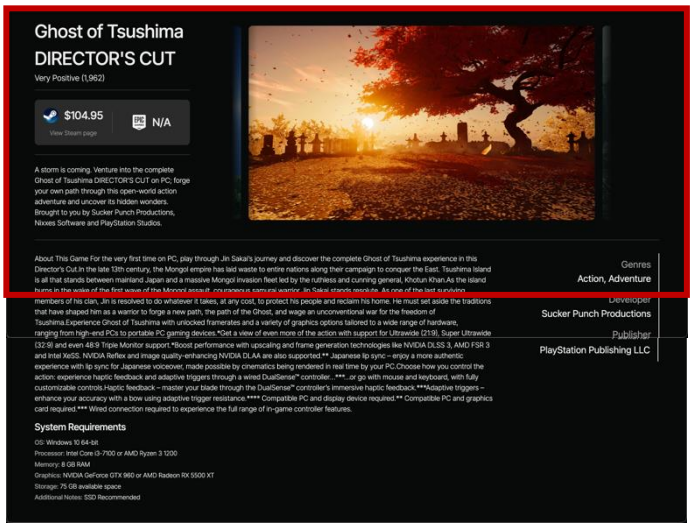
Last button code

Finished pagination menu

Feedback: "I think the game page layout may need some work. Particularly the long description section as having that much text on the screen at once is quite overwhelming"

Improving Game Page Layout

This feedback was also given to me during the Adobe XD designs stakeholder feedback phase, and although I disagreed with it at the time, I had come to realise that it was a valuable critique. Thus, I got to redesigning the game page to ensure that the description text would not overwhelm the user. I knew I wanted the images of the games to be highly prioritised as well as the prices, so these were moved near the top of the page. I wanted my visual hierarchy on the page to be title -> images -> prices -> rest, so I kept this in mind when rearranging the different sections of the page. After some trial and error as well as taking inspiration from both Steam and Epic Games, I came up with the layout to the right. In the red rectangle is landing section of the page (what users can see before scrolling).



With this design, I also had some more room to add more details about the game from the database such as game specifications (system reqs) and a short description (short vs long description are different texts). I also added responsiveness to this page at this point, by simply making a snap point where the content would go from two columns to one and centering all content. To ensure I had improved my design, I asked the stakeholder who gave this feedback as well as two others to judge whether they preferred this new design, or the old one. All three said the new design looked more professional, aesthetic, and seemed more user friendly.

Feedback: “home page feels a bit out of place compared to the rest of the website. I think the images are too big and there isnt enough emphasis on the search bar and text on the left”

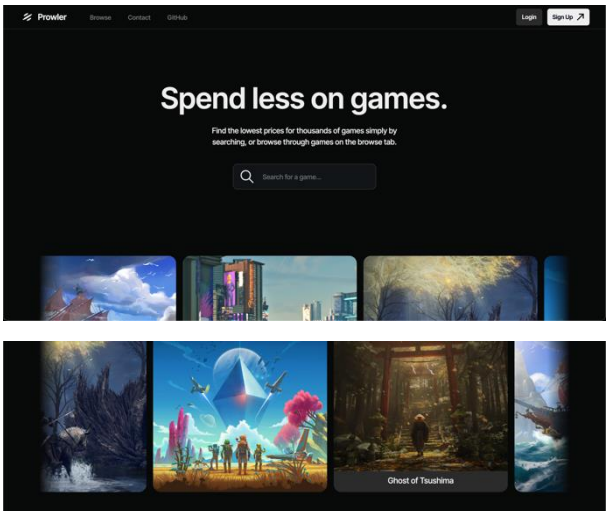
Revamping Home Page Design

Although I did like my home page layout, I agreed with this criticism and felt a redesign of its layout could help improve my website’s aesthetics. On top of this, when the screen shrank the image carousel was only shrinking width-wise meaning the aspect ratio was being changed which is bad practice. I decided to start this process by doing some more research on various websites I liked and how they layed out their landing page. Two websites I used for inspiration were linear[27] and Tailwind CSS’s website[28] because I thought both of their landing pages were sleek, easy to scan, user-friendly, and followed industry standards. One thing these two landing pages had in common was that they both used single-column layouts; a single column layout is beneficial because it simplifies the user experience by creating a clear, focused flow of content.

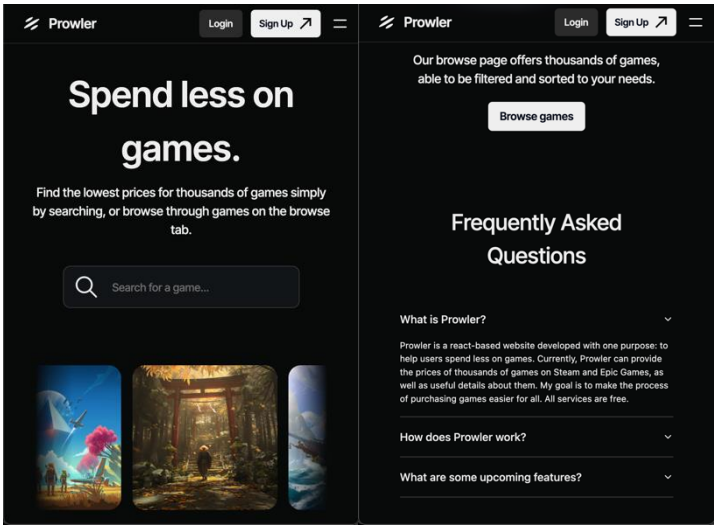
When designing my new home page, another method I integrated from these websites above was their fixed widths at larger screens, as in the width of the content on a page will become fixed beyond a certain point. This means that the page will still be usable on very large screens because I had noticed with my old design that on larger screens, the elements which used % values became much larger than those which were fixed. So, for the parent div of my whole page, I gave it a width of 1320px beyond a screen width of 1360px, and below this width it was w-auto with an mx-5 value. I also applied this method to both other pages, making the page feel more consistent across a range of screens.

---

[27] https://linear.app/
[28] https://tailwindcss.com/



Similar to my games page, redesigning this page was mostly just moving around elements I had already made and changing margins and sizing until I was satisfied. I decided I would put the main element, the main text and search bar, at the top of the column, followed by the carousel of various game's images to establish the visual hierarchy I wanted. During this time, I also switched the carousel from a single image which auto-scrolled (ie move across one image every 2-3 seconds), to three images which auto-played (move through the images at a slow speed at all times). This was very simple due to the component taking the property basis-1/x where x is number of images displayed at any point on the carousel, and using the autoplay option. Again, I made the carousel stop whenever hovered so users could see the game's name and click through to be taken to that game's page.



My final change to the landing page was to add a few less significant but helpful details, as I felt the page felt slightly empty. Since I mention the browse page in the text just below 'Spend less on games.', I added a small message below the carousel with a link to the browse page to guide users through my website. Below this, I used shadcn/ui's accordian component and added an FAQ's section which contained some questions such as 'What is Prowler' for inexperienced users. Although I did plan on having an FAQ's page, I felt that this could be added to home page instead since it wouldn't have the necessary content for its own page. Lastly, for responsivness of the page, all I needed to change was the basis property of the carousel as the screen shrank, going from 1/3 (3 images at a time), to ½ at smaller widths. The rest of the page could easily fit for mobile screens simply by wrapping text and adjusting width values, and thus my home page was complete.

Nearing the end of my allocated development time (it was around Term 3 Week 6 at this point), I made the decision that I would not have sufficient time to add the account functionality I wanted to. My initial plan was to allow users to create and sign into accounts, have wishlists, and receive emails when games in their wishlists went on sale, however looking back this was certainly optimistic considering my limited development time. I decided I would perhaps try to add this in the future, but for now I needed to move on and finish the rest of my site, which from here was only the static contact page.

Contact Page Development

For my contact page, I planned on allowing users to enter their name, email, and message and then submit a form which would be automatically sent to my email address where I could subsequently contact them. I searched up how to make a contact form on your website, and found one of the more popular ways to do this is through third parties like Web3Forms[29] which handle actually sending the email with the user's info to you. I opted to use Web3Forms instead of building my own form submission backend for a few reasons. Firstly,

[29] https://web3forms.com/

using Web3Forms simplified the process significantly, as they handle the backend programs required to send emails. In order to build my own form system, I would need to set up a server, configuring sending emails from the site to my email, and ensuring proper security measures. By using Web3Forms, I could rely on their well-known platform, which is designed with built-in security and privacy measures. This includes encryption of submitted data and protection against spam and bots. Additionally, Web3Forms ensures that all sensitive information, like user emails and messages is handled in a secure way, without me needing to store or process it directly. This not only saved development time but also meant that I didn't need to worry about the security of this system.

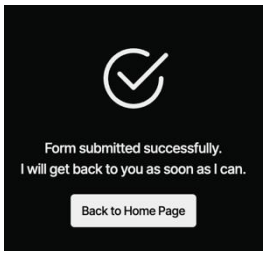
Implementing Web3Forms for User Inputs

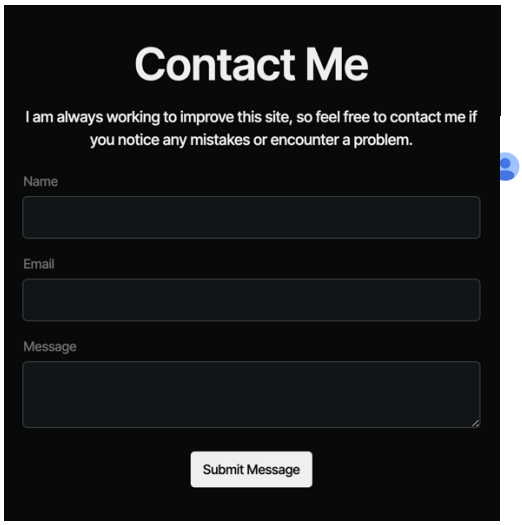
Using Web3Forms was very simple as I just needed an access key from them which I could then input as the 'value' field for my input. Next, I just made a few simple fields for users to fill out: their name, email and message. Subsequently, I created the handleSubmit function which prevents the default form submission behaviour in a React app, allowing it to send form data to the Web3Forms API (which then sends the email to me) without redirecting the user to the Web3Forms website as this would likely be frustrating for users if they wanted to continue using my site. It sets a loading state while awaiting a response and handles success and error cases by updating the state accordingly. This approach ensures a seamless user experience during form submission and allows a small loading icon to be displayed whilst the email was sending.

```
{/* When user submits the form, calls the handleSubmit function */}
<form onSubmit={handleSubmit} className="flex flex-col text-offwhite mx-auto mt-5">
    <input type="hidden" name="access_key" value="fb47546a-eae6-4a69-8e7a-9edad4272b55"/>
```

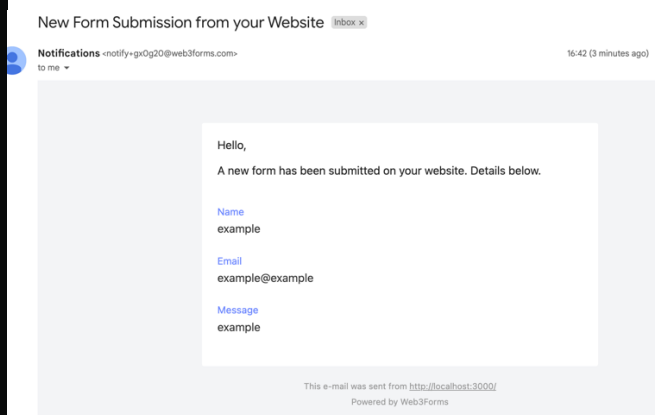
```
// Function for sending off form whilst not redirecting user to a different page
const handleSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    setLoading(true)

    const form = e.currentTarget;
    const formData = new FormData(form);

    try {
        // Fetching response from web3forms
        const response = await fetch("https://api.web3forms.com/submit", {
            method: "POST",
            body: formData,
        });

        // Checking if data recieved successfully and error handling
        if (response.ok) {
            setLoading(false);
            setComplete(true);
        } else {
            console.error("Form submission failed");
            setLoading(false);
        }
    } catch (error) {
        console.error("Error submitting the form:", error);
        setLoading(false);
    }
};
```

Also, when the user clicked the submit button, I noticed they could spam it quickly to send many emails and spam me and potentially overload web3form's system, so I changed the submit button to disappear as soon as it was clicked and replaced with a temporary loader. Then, when the form was submitted, the page would simply display a 'Form Submitted Successfully' message and a home page button to help usability. The final result was a sleek and simple contact page which functioned well. Responsiveness was also very simple due to the page's simple layout; I simply made the fields and text have set left and right margins so that as the screen shrunk, so did they.





Content on contact page pre submission



Email I receive after user submits message

Conclusion

Overall, it's safe to say that I am extremely pleased with the outcome of my project. Despite this, I was still able to identify some next steps to complete in the future.

- Introduce account functionality. This would involve making the sign-up and login forms, setting up authentication to verify user credentials, and securely storing user data.
- Wish list for games. This is interlinked with the account functionality as each user would have their own wish list. Users could add games, and upon these games going on sale would receive an email notifying them.
- Implement language selection for various popular languages such as Chinese, French and Hindi. This would allow my website to become more accessible to users around the globe due to the lack of a language barrier.
- Scrape prices from a larger range of sellers, which would involve setting up a separate scraping file for each vendor. However, it could be highly beneficial; as the number of vendors listed on my website increases, so does the users' incentive to visit and use it.

Final Remarks

My final outcome has far exceeded any of my initial expectations. At the beginning of the year, I thought my website would likely house a couple of hundred games and have some basic filtering functionality; however, it has evolved into a robust platform featuring thousands of games, advanced filtering options, and a user-friendly interface. The site now offers real-time price comparisons from multiple vendors, giving users a seamless experience and making it a valuable resource for gamers seeking the best deals.

My knowledge in technology has also developed considerably. In a little over five months, I have gone from zero experience with React (and Next.js) with only a small amount of JavaScript knowledge to building a full-stack application with a comprehensive backend and carefully thought-out design. This journey has taught me the importance of problem-solving, debugging complex issues, and sticking to specific deadlines. My biggest regret of this process was undoubtedly not having a stakeholder who knew React and had past experience with it. This caused many React specific issues during development where I needed to research them and find similar cases from other people, instead of being able to consult a peer or mentor.

Throughout my development, I was also constantly making choices that aimed to enhance the user experience and align with industry standards. For example, I opted for darker backgrounds on the website as many other gaming related applications do. This is because people typically associate these colours with gaming, and they are easier on the eyes during long browsing sessions. Additionally, I carefully selected the filtering options to include criteria such as price range, genre, and sale status. These filters were chosen based on user feedback and best practices observed in popular game vendors like Steam and Epic Games, ensuring that users can efficiently find the games that suit their preferences.

In terms of functionality, Prowler has also exceeded my initial goals. Users are able to efficiently find data and accurate prices from different vendors for thousands of games on one platform. My data collection method was also automated, lightweight, and extremely quick to run making my data constantly up to date. Users are also able to use the robust filtering system to find games with no problems. Overall, I have sufficiently addressed the problem I laid out in the beginning of this report in the form of a website hosted on Vercel.

Prowler demonstrates how persistence and thoughtful design can turn a simple idea into a useful everyday tool for gamers.