Reece Doyle T3A1

Question 1

```
Provide an overview and description of a standard source control process for a large project
```

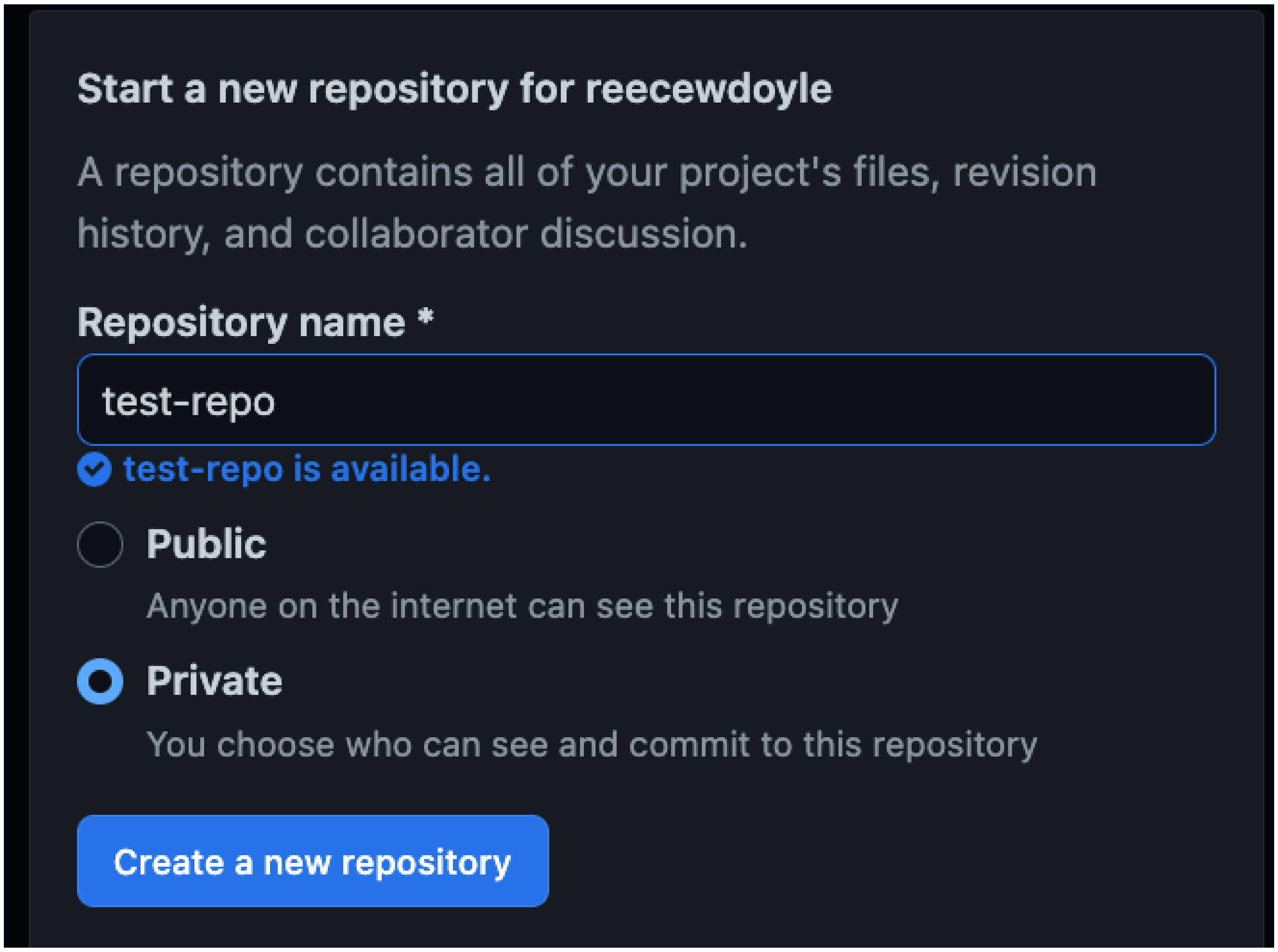
"Provides an extensive overview and description of a standard source control process"

Effective source control of a major project requires all stakeholders to follow a set of protocols and procedures. The specifics may differ between organisations, but as long as everyone within the organisation is following them, the integrity, traceability, and collaboration on the codebase will be maximised.

Repository Setup

A repository needs to be established. This will be where the codebase is stored, shared, and where iterations will be pulled to.

GitHub is one of the most common version control systems due to its ease of use across multiple platforms. Setting up a repository can be as simple as logging into the GitHub website and creating a name that fits the project:



Start a new repository for reecewdoyle

A repository contains all of your project's files, revision history, and collaborator discussion.

Repository name *

test-repo

test-repo is available.

Public

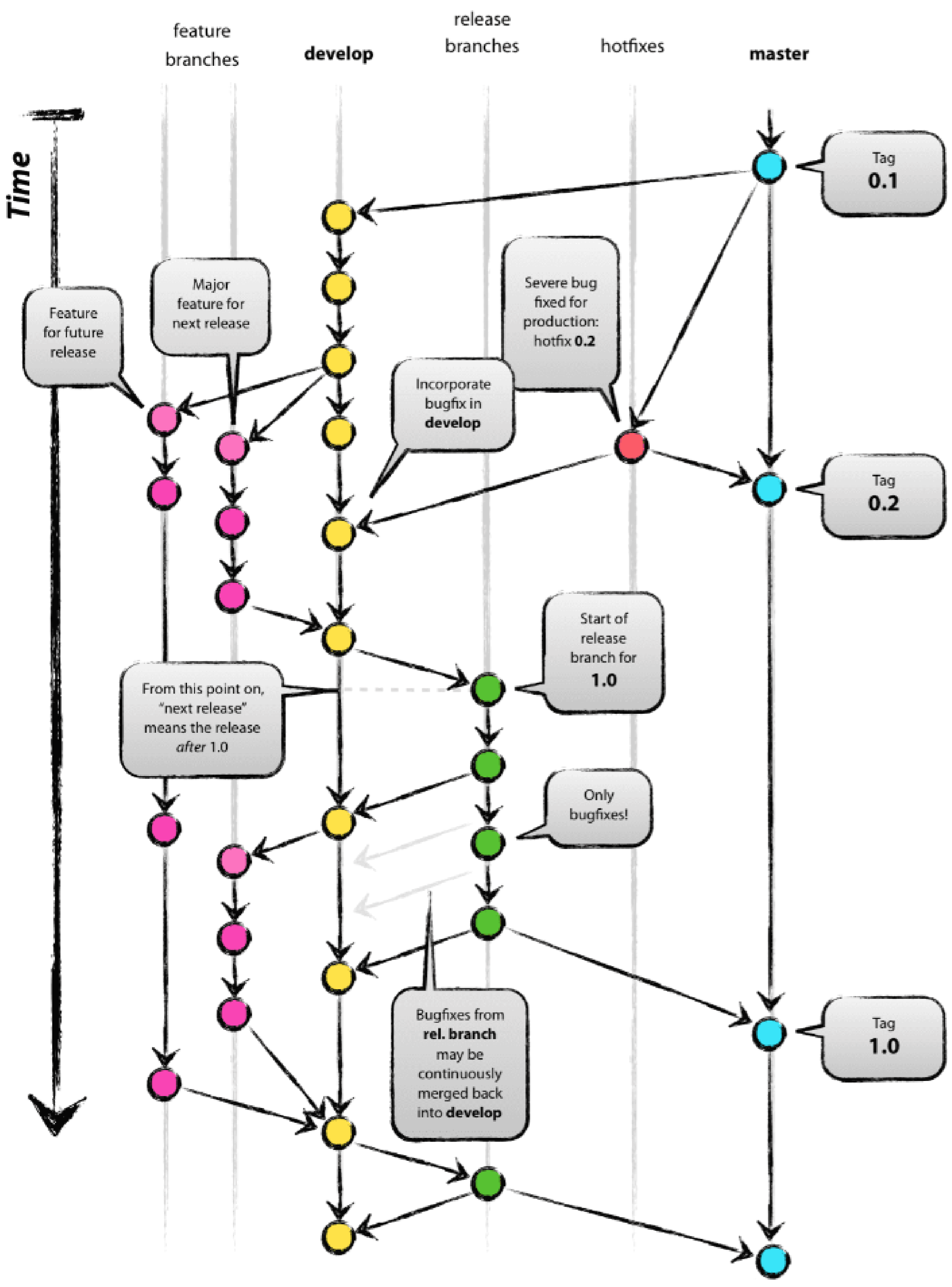
Anyone on the internet can see this repository

[✓ Private]

You choose who can see and commit to this repository

Branching Strategy and Management

The Team Leader will set the expectations for the workflow, but it's likely they will use a branching strategy.


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Major feature for next release
Feature for future release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Feature Branches: This is where individual features can be created and worked on. They are independent from other branches and will generally be one branch per feature. Once a feature is stable enough on its own, it can be merged into the development branch where it can be integrated into the rest of the codebase. In some cases, the feature branch will then be deleted, as it is no longer needed.

Development Branch: This is where features are integrated with the codebase and with each other. Depending on the workflow, the team might move back and forth between the development and feature branches as features are iterated upon.

Release Branch: This is where the codebase is prepared for a new release and receives its final round of testing. Last-minute fixes can happen here without halting ongoing development on the development branch.

Hotfix Branch: This is where quick fixes can be done for critical bugs discovered after the release of the main branch. By this point, the codebase should have gone through many iterations, so major fixes should be rare.

Main Branch: The main (or master) branch is the stable branch that contains the production-ready codebase. This is the point where the project has been developed to a minimum viable product and is ready for release.

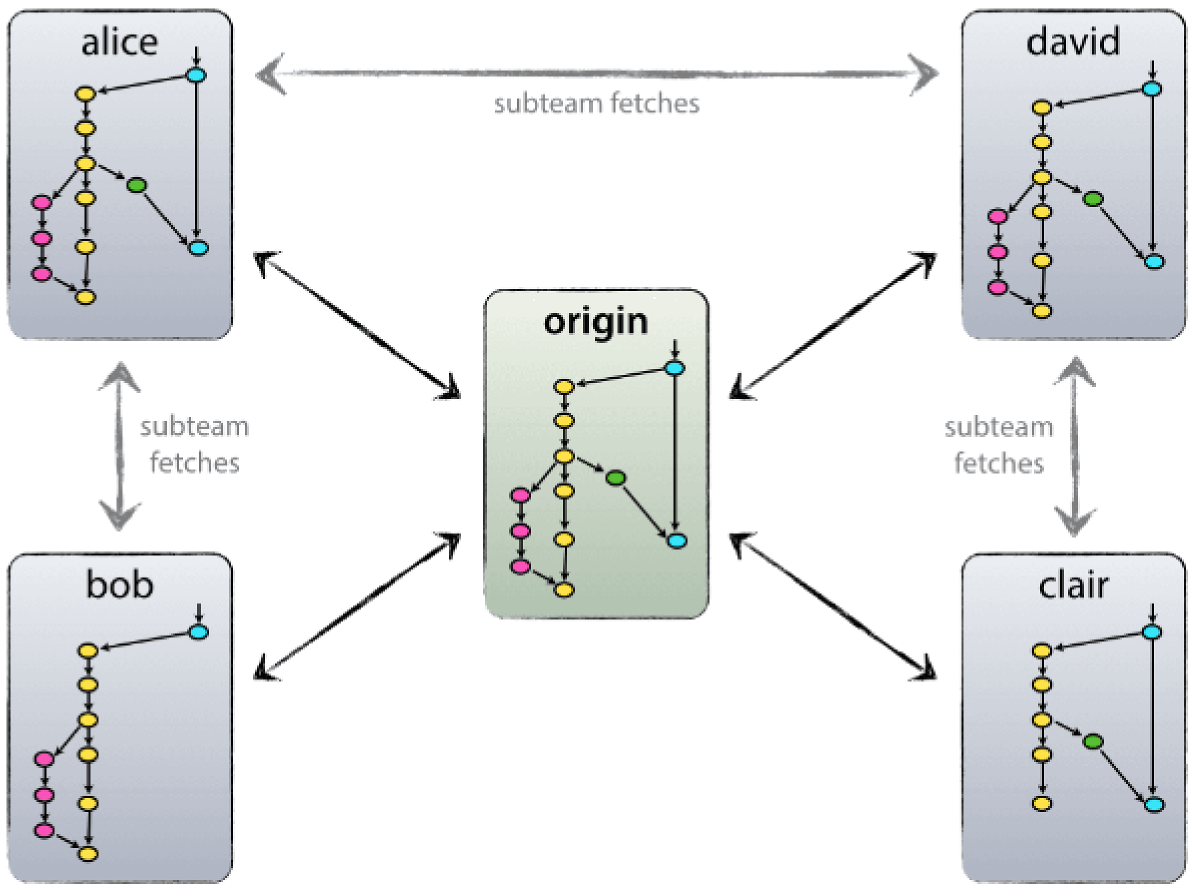
Development Workflow

Managing Team

The Team Leader will delegate tasks for each member. Each member will `fork` or `clone` the Repo that was established to work on using their local machine.

Centralised Repo

Once a feature or task is completed, it is pushed up to the Repo (which in the image below is referred to as `origin`). This is crucial, as each team member could be working on different branches and at different stages of the process at any given time.



Naming Conventions

It's crucial that detailed comments are included with each commit. If it is a fix relating to a ticket generated in GitHub, it is wise to include the ticket number in the commit comment.

conventionalcommits.org suggests a framework where the reason for each commit is clearly stated in the comment:

- `fix`: A commit type of `fix` is designed to patch issues within the codebase.
- `feat`: A commit type of `feat` adds a new feature to the codebase.
- `BREAKING CHANGE` or `!` within the commit type (`fix!`) implies a fix that will involve a change to an API.
- Other types include, but are not limited to, `build`, `docs`, `style`, `refactor`, and `test`.

Push and Pull Requests

Once a developer has completed their work, the Team Leader must first review the code before it is merged into the codebase. Once the work is approved, it can be merged from a feature branch onto the development branch. The same process applies to pull requests.

The Team Leader has the opportunity within GitHub's framework to offer feedback and comments.

Testing

Part of the review process will also include testing, as per Test Driven Development principles. This could be tests that are specifically designed and automated, or manual testing by the Team Leader.

Once the Team Leader is satisfied, the code will be merged into the codebase.

Release Management

A release branch will be created to prepare the product for release. This is where the final level of testing, last-minute fixes, and version tagging can happen (e.g., `v1.0.0`). Once all of this is complete, the release can be merged into the main branch.

Hotfix Management

Any last-minute fixes discovered in the release phase can be addressed on a dedicated hotfix branch. If the development process has been effective, there should be no major issues at this stage. Once the issue has been fixed, it can be tagged (e.g., `v1.0.1`) and merged back into the main branch.

Monitoring and Maintenance

After release, it becomes a continuous improvement process where the team monitors for issues, fixes them as they arise, and develops new features.

References

Atlassian, What is version control?, 2024, https://www.atlassian.com/git/tutorials/what-is-version-control

Atlassian, Feature Branch Workflow, 2024, https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

Driessen, Vincent, A successful Git branching model, 2010, https://nvie.com/posts/a-successful-git-branching-model/

Kiran, Athresh, Step-by-Step Team Workflow with Github Desktop, 2020, https://medium.com/@athresh.kiran/step-by-step-team-workflow-with-github-desktop-

9c1797186c14

conventionalcommits.org, # Conventional Commits 1.0.0, 2024,
https://www.conventionalcommits.org/en/v1.0.0/#summary

Question 2

```
What are the most important aspects of quality software?
```

Much has been written about what makes quality software. Most sources seem to settle on some version of these 6 principles that create quality software.

1. Functionality

This is related to the software's ability to fulfil it's intended purpose. Software should be:

- **Correct:** The software should produce a predictable result. The input should produce the expected output. e.g. Does Quikbooks correctly calculate the tax rates that a business is going to need to set aside?
- **Complete:** All features should be complete. If a feature is not complete, or not implemented, it shouldn't be there.
- **Interoperability:** It's ability to interact with complementary software and third party systems.
- **Security:** Is the user's data safe, and have all security vulnerabilities been handled?

2. Reliability

The best software in the world is only as useful as it is reliable.

- **Availability:** The software needs to be accessible on the optimal platform when needed. e.g. A Map and Directions software that wasn't available on a mobile device would probably not be successful, as the software isn't where you need it to be when you're using it.
- **Fault Tolerance:** The software handles errors gracefully and continues to work when it receives incorrect input. e.g. Does a streaming service have a system in place to withstand a DDoS attack?
- **Recoverability:** The ability for the user's data and the software itself to be recovered quickly.

3. Usability

There needs to be a balance between functionality and usability. A piece of software might be amazing, but if nobody can use the features, it's got a usability problem.

- **Intuitive:** This would be related to the steepness of the learning curve for the end user. If it's too difficult to learn, people won't use it. e.g. if it's a personal finance app, but the interface has too many submenus and the text is too small, people might find it too hard to navigate.
- **Efficiency:** The software must perform a task that allows the user to perform a task quickly. If there are too many steps, the user will abandon the app.
- **Satisfaction:** Using the software should be as pleasant an experience as possible, or users will seek another solution.

4. Efficiency

This could be about how the software integrates into the users system.

- **Performance:** This is to do with the speed and responsiveness of the software. It must be quick to perform tasks and move between screens. e.g. Web based apps must be able to withstand high traffic to remain responsive.
- **System Resources:** This is the amount of system resources the software requires. This could relate to the CPU, the size of the files, and the size of the system required to perform optimally.
- **Scalability:** This is can be the make-or-break point for a piece of software. The performance must justify the expense of integrating it into the system or it won't be viable. e.g. Cheaper and worse performing software can still be selected over the more expensive software when the financial outlay doesn't stack up.

5. Maintainability

How easily the software can be maintained and updated.

- **Modularity:** A modular piece of software would be made of seperate parts that can be independently modified, fixed and upgraded. e.g. if a codebase is written in a single large file, it will be very difficult to find and isolate a problem.
- **Reusability:** Can the code be reused for multiple purposes?
- **Analysis**: The ease at which the software can be analysed to determine defects and vulnerabilities.

6. Portability

This is the software's ability to be transferred from one environment to another.

- **Adaptability:** The ease with which the software can be adapted to different environments. e.g, will the software work on Windows, Linux, and Mac?
- Installation: The ease at which the software can be configured within a system.

References

Croft, Andi, 6 Essential Metrics for Software Quality, 2022, https://huddle.eurostarsoftwaretesting.com/6-essential-metrics-for-software-quality/

Jayalakshmi Iyer, Sanika, What is Software Quality Assurance, and why is it important?, 2023, https://www.turing.com/blog/software-quality-assurance-and-its-importance/#:~:text=The%20three%20primary%20aspects%20of,reliability%2C%20portability%2C%20and%20efficiency.

Cepta Infotech, 6 Key Aspects of Testing Quality Software and Types, 2020, https://www.cetpainfotech.com/blogs/6-key-aspect-of-testing-quality-software-and-types

---

Question 3

```
Outline a standard high level structure for a MERN stack application and
explain the components
```

"Shows almost flawless understanding of the high level structure of the app"

MERN Stack Applications

A MERN stack application if a full stack Web Application that utilises:

- **MongoDB:** A non- Relational Database to store application data,
- **Express:** a Node.js server-side framework for building APIs,
- **React:** a JavaScript Frontend library for building user interfaces.,
- **Node.js:*** an open-source server-side application using JavaScript.

The MERN Stack utilises the Model-View-Controller architecture pattern to create a Full stack Web Application.

Model

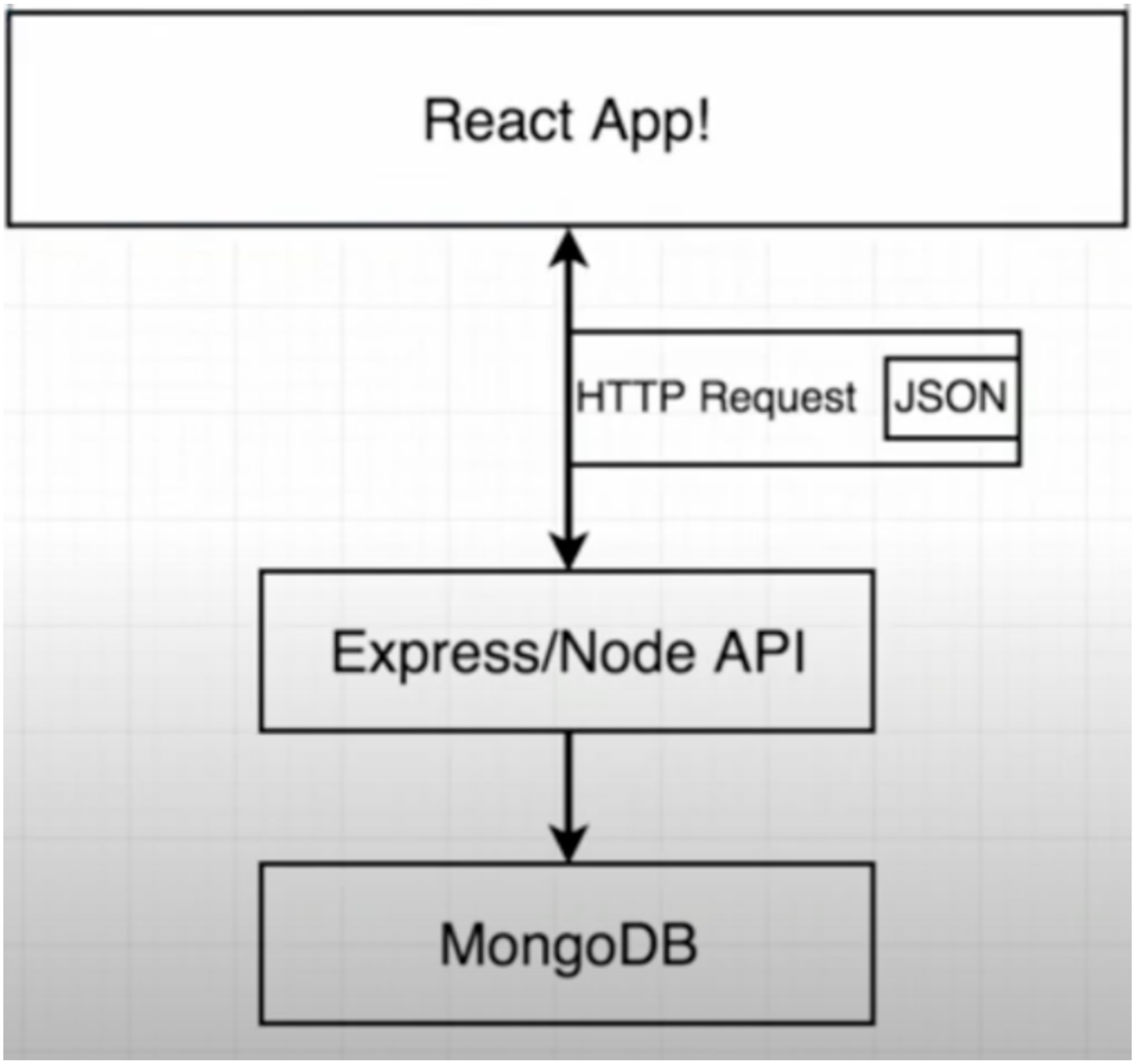
This component of the app is responsible for managing the storage and retrieval of data. In a MERN stack, the database used for this is MongoDB. The model section of the project will contain schemas that dictate how the rest of the app interacts with the database.

View

This component of the app is responsible for how the data is presented to the end user. In a MERN stack, this is handled by React.js. This is the frontend framework where the User Interface (UI) is created. This is where the user can create their queries to send to the rest of the app.

Controller

This component of the app is responsible for handling user data and updating the Model and View components accordingly. In a MERN stack, this is handled by Node.js and Express.js.

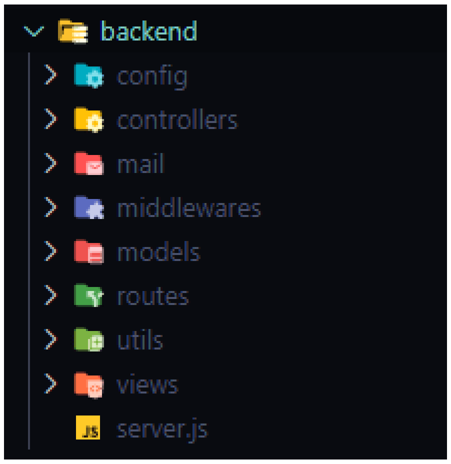


React is a JavaScript library used to build the User Interface (UI). The application communicates with a **Node.js API** using HTTP requests in JSON format. The API accesses a **MongoDB** database to return the results of the queries of the user.

Structure

To create a quality application (as per the six principles of quality software mentioned above), it makes sense to split the frontend and backend into separate directories.

Backend Directory



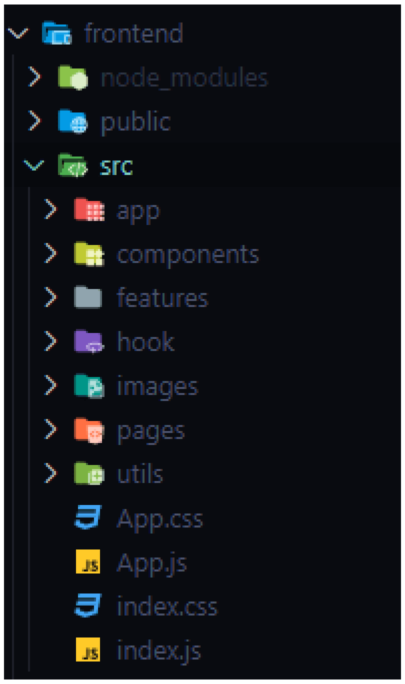
```
backend
  config
  controllers
  mail
  middlewares
  models
  routes
  utils
  views
  server.js
```

This is an example of a server directory where the backend code is built using Node.js and Express.js.

Each section of the code is broken up into files and folders to make it easy to find and maintain:

- **config:** This is where configuration files will be.
- **controllers:** This is where the controller files will be. They handle HTTP requests and responses.
- **models:** This is where the models will be. This handles how the application interacts with MongoDB.
- **routes:** This is where the routing for the application is detailed. This determines how each file is connected to each other.
- **server.js:** This is the entry point for the backend server.

Frontend Directory



```
frontend
  node_modules
  public
  src
    app
    components
    features
    hook
    images
    pages
    utils
    App.css
    App.js
    index.css
    index.js
```

This is an example of a frontend file directory built using React, which works with the above backend directory.

Each section of the code is broken up into files and folders to make it easy to find and maintain:

- **public:** This is where the HTML files that determine how the website will be displayed to the public are kept.
- **src:** This contains the source code for the frontend application. The `src` folder must contain the following:
  - **components:** This is where the React components are found. They are what build the UI.
  - **app.js:** This is the file where the frontend application renders components of the web app.
  - **index.js:** This is the entry point to the frontend of the app.

References

geeksforgeeks.org, MERN full form, 2024,
https://www.geeksforgeeks.org/mern-full-form/?ref=lbp

geeksforgeeks.org, MERN stack, 2024,
https://www.geeksforgeeks.org/mern-stack/?ref=lbp

Amankwah, Kingsley, MERN Stack Project Structure: Best Practices, 2023
https://dev.to/kingsley/mern-stack-project-structure-best-practices-2adk

---

Question 4

```
A team is about to engage in a project, developing a website for a small
business. What knowledge and skills would they need in order to develop the
project?
```

"Effectively describes a range of skills and knowledge required by IT workers to complete a quality web development project"

Designing a website for a small business is a very broad topic that could be taken in many different directions. In relation specifically to the skills and knowledge required of a developer or team to deploy such a website, it would be assumed that the following had already been decided by the client or team leader:

- The purpose of the website,
- The Domain and host,
- The design elements, sketched out in a wireframe, with colours, possibly on Figma, specifying in as much detail as possible how many pages are required, what will be on them, and how they will be mapped,
- Copy for the website,
- Assets required for the website (logos, pictures, videos and fonts.)
- Deadlines and milestones to complete the project.
- The organisational method for meeting those milestones (possibly Agile with a Kanban board. Trello etc.)
- Clearly defined tasks and responsibilities broken down amongst team members.

A small business website will require team members with both Front End and Back End skills. This could be two dedicated people, or possible a single person. Depending on the size of the project, the size of the team would be scaled accordingly.

Front End Skills

The Front End is the part of the website that is user facing. The part that looks pretty!
In order to create a successful User Experience, the Front End developer would require some level of proficiency in:

1. HTML/CSS (Hyper Text Markup Language & Cascading Style Sheets): These are the languages that determine what the structure and styling of the website is going to look like. HTML deals mostly with the the structure, and as the name would indicate, CSS deals with the styling.
2. JavaScript: This is the programming language used in conjunction with HTML and CSS used to make dynamic web applications.
3. Responsive Design: This is to determine how well the website works on different devices. Will the website work if it is viewed on a phone? As mobile devices have become far more prevalent than desktop computers, Mobile First Design has become the way many Developers approach their designs.
4. Cross-Browser Compatibility: Your website might look good in Chrome, but it also needs to work on Firefox, Safari, and Microsoft Edge. Developers need to be sure that their code works consistently across all the major web browsers.
5. Design skills: While a Web Dev might not be expected to be the person to design a website, it's inevitable that some design may come up along the way. This could be a feature that the client has missed, or possibly that something that isn't working. It would be beneficial for a Web Dev to have some ability to create their own designs to help communicate with the client.

Back End Skills

While the Front End makes the Web App usable and aesthetically pleasing, the Back End is the "plumbing" and "boiler-room" that actually makes the Web App work. This is the part that makes the User Experience possible. You will need some proficiency in:

1. Programming Languages: This could be Java, Python, Ruby, C# or many others that run server-side applications that manage the data that is being collected, called and displayed.
2. Databases: This could be PostgreSQL, MySQL, Oracle, MongoDB or some other kind of Relational Database. The Back End Dev will need to know how to query, store and retrieve data effectively.
3. Web-servers: Back End Devs need a good knowledge of server side scripting and how to configure and manage servers. This could be Apache or Ngnix.
4. APIs (Application Programming Interfaces): Back End Devs must be able to build and use APIs, which are used as a means of communicating between different systems. e.g. Connecting a PostgrSQL database to a website, and allowing the user the ability to Create, Read, Update and Delete entries on the Database.

Common Skills to both Front End and Back End Web Devs

Front End Dev and Back End Dev responsibilities can be split across two different people to take advantage of the strengths of each. Those who are strong with UI/UE (User Interface/User

Experience) might not be as strong with Back End, and vice verse. Some developers might choose to specialise in Front End or Back End, or work within either specialised role in a larger project.

Some of the common skills that both kinds of Dev need to have are:

1. Version Control: Devs need to be masters of a version control like Git and GitHub to allow for a successful workflow, regardless of whether or not they're working by themselves or within a team. This allows for the code to be backed up, reviewed, iterated and improved all from a single place. it solves the issues that can come up with multiple people working on a project, but also is a way of backing up data from the local machine.
2. Problem Solving: It almost goes without saying that problem solving is one of the most important soft skills that any kind of Web Dev needs to master in order to be effective. An effective Dev will be able to problem solve their way out of most situations. You need to be able to reach into the large repository of knowledge that is available on forums like Stack Overflow, as it's highly likely that someone else has experienced the problem you're having before. The integration of AI into many workflows now means that most workers (not just IT) will have to become very good at prompting AI to speed up (but not replace) their workflows.
3. Communication: Devs need to be able to effectively communicate with all stakeholders of the project to have the deliverables ready on time. This should be an incremental process where communication happens at regular intervals. This could be in the form of a Standup meeting, within emails, face-to-face, within GitHub for code reviews from the Head Developer or even across organisations. There should be some clear guidelines for where communication occurs, and when it happens. There's nothing worse than having to dig through an email chain only to find that you've been working on the wrong feature, or solving a problem that's already been solved by someone else.
4. Testing and Debugging: While this might more commonly be associated with Back End Devs, where Test-Driven Design dictates that the first element of a design is the the test to prove that a function works, this is also apart of a Front End Dev's work day. Coding should be created in such a way that it can be tested, and is labeled with as many comments as necessary such that a Dev that has never seen the code can figure out how it works, or that the Dev who wrote it can come back years later and pick up where they left off.
5. Security: A good knowledge of vulnerabilities within a design that will make a website more susceptible to cyber attacks minimises the downtime and helps to meet the obligations the vendor has when handling the data of their customers.

developing a website for a small business.
Possibly look at the marriage celebrant website.

References

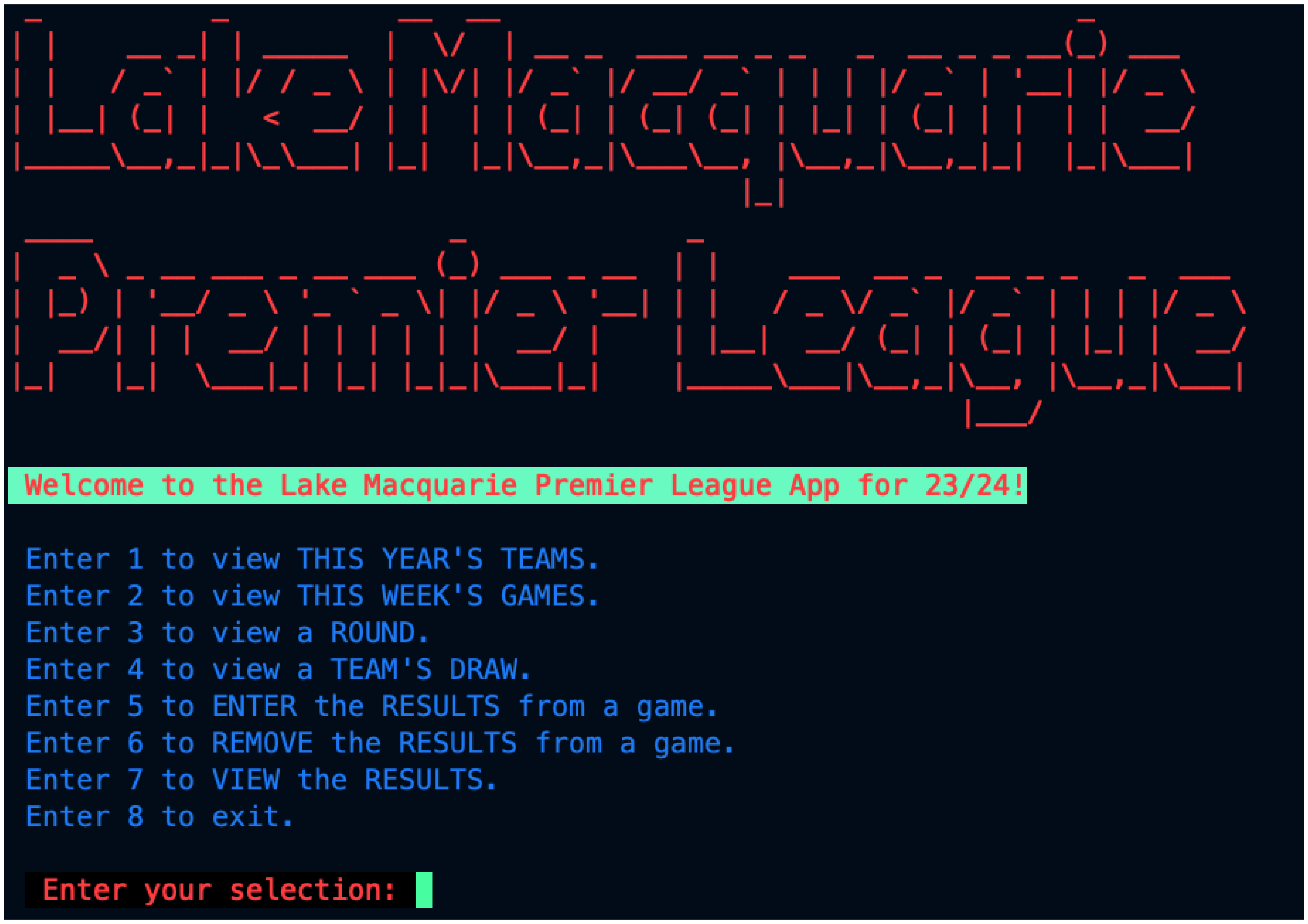
Smith, Corey, What skills do you need to build a website?, 2017, https://www.tributemedia.com/blog/what-skills-do-you-need-to-build-a-website

Question 5

```
With reference to one of your own projects, discuss what knowledge or skills
were required to complete your project, and to overcome challenges
```

CMP1043-3.2 Within your own project what knowledge or skills were required to complete your project, and overcome challenges?

"Effectively describes a range of skills and knowledge used to complete a project."



```
Lake Macquarie
Premier League

 Welcome to the Lake Macquarie Premier League App for 23/24!

 Enter 1 to view THIS YEAR'S TEAMS.
 Enter 2 to view THIS WEEK'S GAMES.
 Enter 3 to view a ROUND.
 Enter 4 to view a TEAM'S DRAW.
 Enter 5 to ENTER the RESULTS from a game.
 Enter 6 to REMOVE the RESULTS from a game.
 Enter 7 to VIEW the RESULTS.
 Enter 8 to exit.

  Enter your selection:
```

My first command line app was the Lake Macquarie Premier League app, which was written in Python for the T1A3 assignment.

The Lake Macquarie Premier League app is designed to be used by an administrator of an amateur Football (Soccer) competition in Lake Macquarie, NSW, Australia.

It is designed to efficiently handle data for a 14 Round season among 8 local teams. Each team plays the 7 other teams in the competition twice per year, once at their home ground, and once as the away team at their opponents home ground, for a total of 56 games, played from November 2023 to March 2024.

The goal was to make the data for the entire season easy to store, and retrieve at a moments notice, based on a few predicted user stories:

- The administrator who wants to be able to store results in a simple CSV that could then be used to feed an Excel Spreadsheet.
- The fan, who wants to how many teams are in the competition and who they are. They will also be able to look up the games occurring in the coming week, or many weeks into the future. The may also want to check games that have already happened.
- The player, who wants to be able to plan their life around the games and wants to access easily where they'll be playing for the whole season in advance, and also look back at where they played.
- Any of the above users who want to be able to view a running tally of results of the games throughout the season.

To meet all of these user stories, I decided that their would need to be 7 features in total:

1. Teams Competing
2. Games this week
3. Display Rounds
4. Display the Draw for a particular team
5. Enter Results of a game
6. Delete Results of a game
7. View the current results in a table

Skills

At a more Macro level, most of the skills I had to develop to complete this task were:

- Python Syntax
- How to import from packages and other python files
- The Syntax for the packages I imported
- While loops,
- Try and Except Blocks
- Testing

Analysing the code from the home screen is probably the most efficient way of getting to each of the skills required to develop this app.

```
from colored import fore, back, attr
from lmpl_functions import (
    print_teams, print_this_round, print_round, team_draw,
    enter_results, edit_results, view_results
)
import pyfiglet as pfg
import csv

# App Logo
text = pfg.print_figlet(text = "Lake Macquarie Premier League", font =
"standard", colors = "red")

# App Greeting
print(f"{fore('red')}{back('white')} Welcome to the Lake Macquarie Premier
League App for 23/24!{attr('reset')}\n")

file_name = "results.csv"

try:
    # open the file in read mode
    results = open(file_name, "r")
    results.close()
    # print("In try block")
    # if it throws an error, the file doesn't exist
    # if no error, the files exists
except FileNotFoundError:
    # Now, we know the file doesn't exist
    # Create a new file
    results = open(file_name, "w")
    results.write("Game Number,Home Team,Home Goals,Away Team,Away Goals\n")
    results.close()
    # print("In except block")

def create_menu():
    print(f"{fore('blue')} Enter 1 to view THIS YEAR'S TEAMS.
{attr('reset')}")
    print(f"{fore('blue')} Enter 2 to view THIS WEEK'S GAMES.
{attr('reset')}")
    print(f"{fore('blue')} Enter 3 to view a ROUND. {attr('reset')}")
    print(f"{fore('blue')} Enter 4 to view a TEAM'S DRAW. {attr('reset')}")
    print(f"{fore('blue')} Enter 5 to ENTER the RESULTS from a game.
{attr('reset')}")
    print(f"{fore('blue')} Enter 6 to REMOVE the RESULTS from a game.
{attr('reset')}")
    print(f"{fore('blue')} Enter 7 to VIEW the RESULTS.  {attr('reset')}")
    print(f"{fore('blue')} Enter 8 to exit.\n")
    choice = input(f"{fore('red')} {back('16')} Enter your selection:
{attr('reset')}" "")
    return choice

users_choice = ""

while users_choice != "8":
    users_choice = create_menu()
    if (users_choice == "1"):
        print_teams()
    elif (users_choice == "2"):
        print_this_round()
    elif (users_choice == "3"):
        print_round()
    elif (users_choice == "4"):
        team_draw()
    elif (users_choice == "5"):
        enter_results(file_name)
    elif (users_choice == "6"):
        edit_results()
    elif (users_choice == "7"):
        view_results()
    elif (users_choice == "8"):
        continue
    else:
        print("Invalid Input")
```

Imports

First I had to learn how to `import` properly. Rather than just having the app in a single file, I broke it into several: `main.py`, `lmpl_functions.py` and `test_lmpl_fucntions`. Each of the functions that are selectable from the home screen are in the `lmpl_functions` file and listed as `imports` below.

```
from lmpl_functions import (
    print_teams, print_this_round, print_round, team_draw,
    enter_results, edit_results, view_results
)
```

I also had to import these 3 packages: `colored`, `pyfiglet`, and `csv`.

```
from colored import fore, back, attr
import pyfiglet as pfg
import csv
```

Colored

Learning to use `colored` allowed me to change the colours of the text in my app. This was especially helpful with readability. It also allowed me to make the team names reflect the team colours in the `print_teams`, `team draw`, and the `enter_results` functions.

`fore, back` allowed me to make the Greeting and the menu selections have a different background and text colour.

```
# App Greeting
print(f"{fore('red')}{back('white')} Welcome to the Lake Macquarie Premier
League App for 23/24!{attr('reset')}\n")
```

This gave my greeting red text and a white background.
NB: the background appears green in my screenshot above because I'm using the `powerlevel10k` theme from `Oh My Zsh` on my local machine.

I used `colored` for the menu selections on the homepage to make the Blue as well:

```
print(f"{fore('blue')} Enter 1 to view THIS YEAR'S TEAMS. {attr('reset')}")
```

PyFiglet

`Pyfiglet` was another package I had to learn to use to create the logo. I believe `Pyfiglet` is a version of a more popular package called `figlet`, optimised for Python. The syntax was rather simple, and allowed me to create the red `Lake Macquarie Premier League` logo above with only that single line of code. I also used it to make the exit screen logo, which simply reads `L.M.P.L Thank You!`

```
import pyfiglet as pfg
# App Logo
text = pfg.print_figlet(text = "Lake Macquarie Premier League", font =
"standard", colors = "red")
```

CSV

Most of this app relies on manipulating and viewing the data in a CSV file, therefore it was crucial to learn to use `csv`.

```
file_name = "results.csv"

try:
    # open the file in read mode
    results = open(file_name, "r")
    results.close()
    # print("In try block")
    # if it throws an error, the file doesn't exist
    # if no error, the files exists
except FileNotFoundError:
    # Now, we know the file doesn't exist
    # Create a new file
    results = open(file_name, "w")
    results.write("Game Number,Home Team,Home Goals,Away Team,Away Goals\n")
    results.close()
    # print("In except block")
```

When the app is activated the first thing it does is look for a file called `results.csv`. If it exists, it opens the file in `read mode`, invoked by `results = open(file_name, "r")` and then closes it with `results.close()`.

If the file doesn't exist, it creates the file with `results = open(file_namem "w")`, writes the column names with `results.write("Game Number,Home Team,Home Goals,Away Team,Away Goals\n")` and then closes the file with `results.close()`.

Try and except blocks

The Macro skill I had to learn above this was also `Try` and `Except` blocks and error handling. Without this, the app would throw errors when it didn't find the `results.csv` file and crash.

While Loops

Another skill I had to learn was how to make `while` loops. This was used for the selection menu.

```
users_choice = ""

while users_choice != "8":
    users_choice = create_menu()
    if (users_choice == "1"):
        print_teams()
    elif (users_choice == "2"):
        print_this_round()
    elif (users_choice == "3"):
        print_round()
    elif (users_choice == "4"):
        team_draw()
    elif (users_choice == "5"):
        enter_results(file_name)
    elif (users_choice == "6"):
        edit_results()
    elif (users_choice == "7"):
        view_results()
    elif (users_choice == "8"):
        continue
    else:
        print("Invalid Input")
```

First I had to establish the variable of `users_choice` having a value of `""`, which is essentially nothing. This was so we could call it in the `while` loop.

decided that `8` would be what I used to exit the menu, so I used the logic `while users_choice != "8":` to mean "while the users choice isn't 8". A selection of `8` would send you into the `elif` block of `elif (users_choice == "8"): continue`. The `continue` would send you into a print function at the end of the app, once again using `pyfiglet` to create a farewell:

```
# App Fareweall and Logo

print("\n")

print(f"{fore('red')} {back('white')} Thank you for using the Lake Macquarie
Premier League App! {attr('reset')}")

print("\n")

text = pfg.print_figlet(text = "Thank You !\n\nL . M . P . L", font =
"standard", colors = "red")
```

The rest of the `elif` loops would invoke the respective functions.

In order to error handle, I had the `else: print("Invalid Input")`, meaning anything other than a number from 1-8 would send the user into the else loop.

Date Time

On the functions page, I used Python's built in `datetime` package to solve a problem I was having. I wanted to create a function that printed only the games that were occurring in that particular week. I wanted the user to be able to look on a Tuesday what games were being played on the weekend.

As I'd already defined the games by game numbers and dates, the most obvious way to create a grouping that would print easily was to use `week_num` in `datetime`.

First it would print the date , then use `isocalendar` to determine the week of the year. Each week of the year has a week number (first week of the year is week 1, last week of the year is week 52, sometimes 53 if it's a leap year).

```
def print_this_round():
from datetime import date

# Show actual date

my_date = datetime.date.today()
today = date.today()
print(today.strftime(f"\nToday's date is %A %d %B %Y"))

# Using isocalendar() function

year, week_num, day_of_week = my_date.isocalendar()
print(f"\nIt is week #" + str(week_num) + " for the year\n")
print("This week's games are: ")
found_games = False # Variable to track if any games are found

with open("lmpl.csv", "r") as f:
reader = csv.reader(f)
for row in reader:
if row[0] == str(week_num):
print("\n" "Round " + row[1] + "," + " " + "Game" + " " + row[2] + "," + " "
+ row[3] + " " + "vs" + " " + row[4] + " " + row[5] + "," + " " + row[6] +
"," + " " + row[7])
print("\n")

found_games = True # Set to True if at least one game is found

if not found_games and week_num in range(6, 43):
print("No Games this week.")
```

Once the `week_num` was determined, the week of the year would be printed, as long as the `week_num` was in the range of 6 and 43. This was because the competition ran from week 44 of the year to week 5 of the following year.

I stored the draw for the competition in a seperate csv file called `lmpl.csv`. This meant I had to add an extra column to my `lmpl.csv`, but it was well worth it for this functionality:

```
Week,Round,Game Number,Home Team,Away Team,Day,Date,Time
44,1,1,Belmont Bandits,Boolaroo Bulldogs,Friday,3/11/23,18:00
44,1,2,Charlestown Cobras,Eleebana Eagles,Saturday,4/11/23,12:00
44,1,3,Glendale Guardians,Speers Point Spartans,Saturday,4/11/23,15:00
44,1,4,Swansea Silverbacks,Warners Bay Wanderers,Sunday,5/11/23,14:00
45,2,5,Boolaroo Bulldogs,Charlestown Cobras,Friday,10/11/23,18:00
45,2,6,Eleebana Eagles,Glendale Guardians,Saturday,11/11/23,12:00
45,2,7,Speers Point Spartans,Swansea Silverbacks,Saturday,11/11/23,15:00
45,2,8,Warners Bay Wanderers,Belmont Bandits,Sunday,12/11/23,14:00
46,3,9,Belmont Bandits,Charlestown Cobras,Friday,17/11/23,18:00
46,3,10,Boolaroo Bulldogs,Eleebana Eagles,Saturday,18/11/23,12:00
46,3,11,Glendale Guardians,Swansea Silverbacks,Sunday,19/11/23,14:00
46,3,12,Speers Point Spartans,Warners Bay Wanderers,Friday,24/11/23,18:00
```

I once again used the `csv reader` to look at the `lmpl.csv` and print out the games with that corresponding `week_num` in the format below:

```
reader = csv.reader(f)
for row in reader:
if row[0] == str(week_num):
print("\n" "Round " + row[1] + "," + " " + "Game" + " " + row[2] + "," + " "
+ row[3] + " " + "vs" + " " + row[4] + " " + row[5] + "," + " " + row[6] +
"," + " " + row[7])
print("\n")
```

Testing

As most of my functions were reading and writing based, they were difficult to write tests for. However, I was able to create a test for the above `print_this_round` function.

As I was using real time data, I had to come up with a test that would force the function to throw the negative `No Games this week` print statement.

To do this, I simply copied the code, commented out what I didn't need, and declared `week_num = week_num_to_test`. I then declared `week_num_to_test = 10`. This meant the function stayed as intact as humanly possible. As 10 is outside of the range of 6 and 43, it printed `No Games this week`. Considering this project was delivered in late December 2023, the only way to prove this function had been error handled gracefully would've been to wait until late February 2024. Obviously not ideal.

```
import csv
# import datetime

def print_this_round():
        # from datetime import date
        # # Show actual date
        # my_date = datetime.date.today()
        # print(my_date)
        # # Using isocalendar() function
        # year, week_num, day_of_week = my_date.isocal`endar()
        # print("Week #" + str(week_num))
        # print("This week's games are: ")

        week_num = week_num_to_test

        found_games = False # Variable to track if any games are found

        with open("lmpl.csv", "r") as f:
                reader = csv.reader(f)
                for row in reader:
                        if row[0] == str(week_num):
                                print("\n" "Round " + row[1] + "," + " " +
"Game" + " " + row[2]
                                                 + "," + " " + row[3] + " "
+ "vs" + " " + row[4] + " "
                                                 + row[5] + "," + " " +
row[6] + "," + " " + row[7])
                        print("\n")
                        found_games = True # Set to True if at least one
game is found

        if not found_games and week_num in range(6, 43):
                print("No Games this week.")

# Test the function with a specific week_num value
week_num_to_test = 10 # Change this to the desired week number
print_this_round()
```

Question 6

```
With reference to one of your own projects, evaluate how effective your
knowledge and skills were for this project, and suggest changes or
improvements for future projects of a similar nature
```

"Evaluates effectiveness of knowledge and skills accurately, providing examples, and providing an insightful improvement on each skill"

My Term 2 Flask app started with the best of intentions......

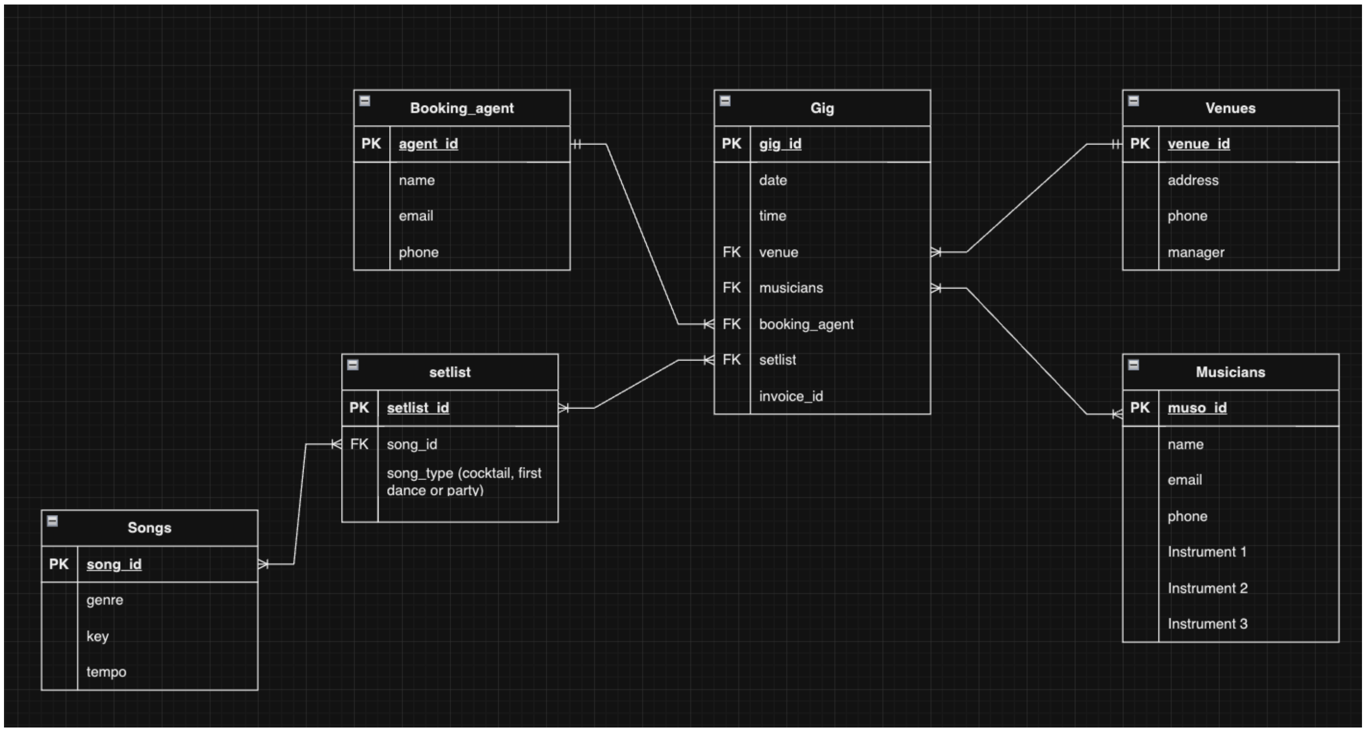
I've worked as a performing musician for hire since I was 17. I've done hundreds of gigs in numerous settings, but almost always playing covers for events, parties, or in pubs and clubs.

Over time I've gone from being a "fill-in" musician working with whoever will have me to running my own bands. Working for other people allowed to see how my friends and colleagues ran their bands. I picked the best parts of what they did and iterated them into my own policies and procedures.

Weddings are my favourite kinds of gigs to do, because you get to be a part of one of the most important days in a couple's life. There's always a crowd. There's always a vibe. And the pay is great, but it's a very high stakes gig and I've played in some train-wrecks...(other peoples gigs. Not mine of course.....) which can be catastrophic for a wedding vendor. All it takes is one bad review to potentially sink your business.

My goal was to create an app that would allow me to streamline the many tasks a wedding band manager/musician would likely encounter in the planning phase, thus maximising our ability to deliver a high quality performance on the day.

All wedding gigs have essentially the same elements that fit nicely into a relational database. This was my initial ERD:



The goal was that I would be able to record all the data needed for the gig in a relational database. This included the venue, booking agent, the musicians playing, the setlist we were going to play, and the function of that setlist (cocktail hour, first dance, or party). I was going to then have a list of songs in a database that would allow us to store the information I needed for each song. This would correspond to a collection of songs I would have PDF charts for.

All paths flowed back to the "gig" itself. By looking up a single gig entry, I could everything I needed to know about a given gig.

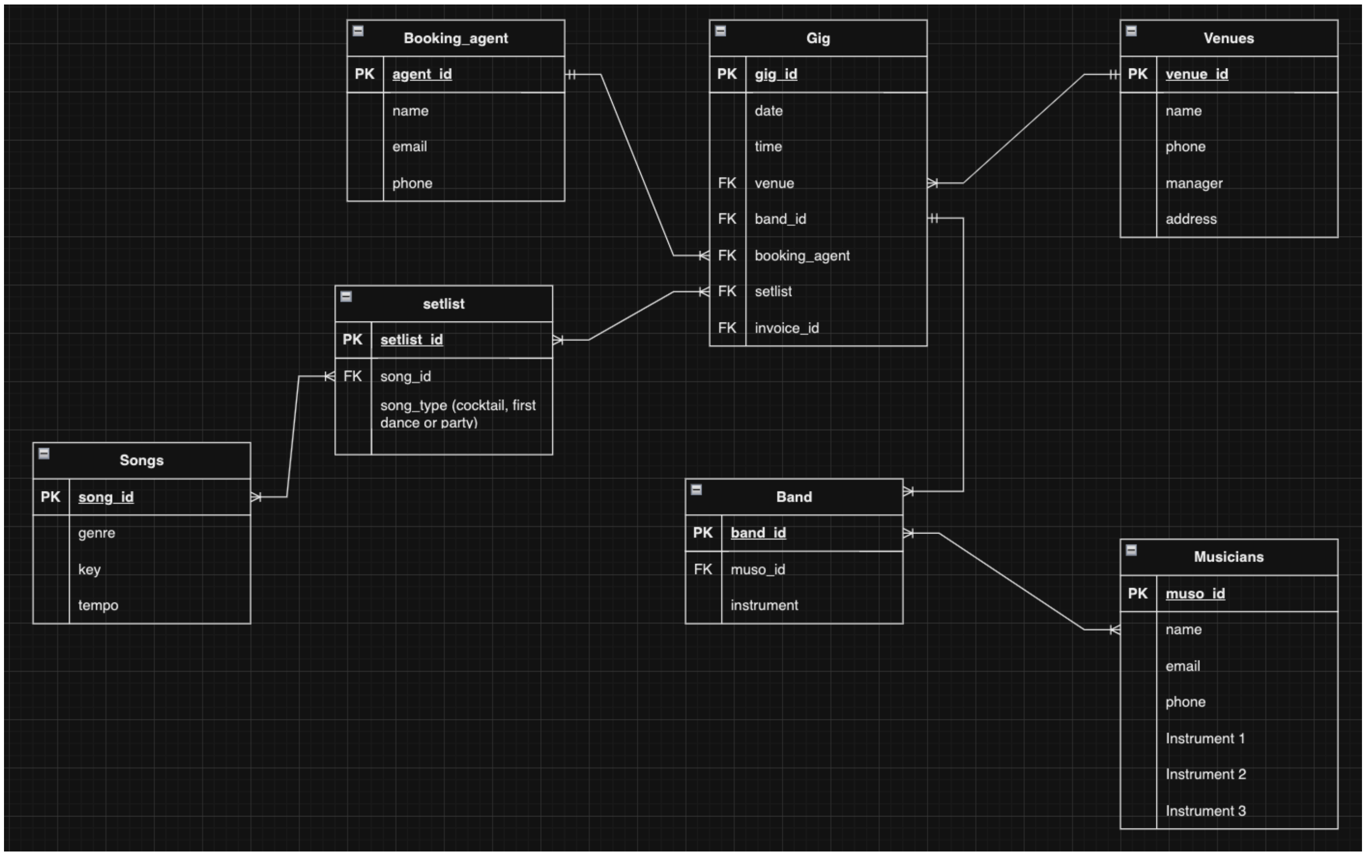
One of the major problems I encountered when I actually went about building this API is that I had a mistake in the relationship between the `setlist` and `songs` tables. This is a Many-to-Many relationship which required a join table between them.

I can now say from experience that join tables are much harder to make work than I anticipated, and were well above my skill set at the time (or even now for that matter, as I haven't revisited them!).

On review of the lecture recordings I watched to try and work out why I couldn't make the join tables work, I noticed Simon mention that he rarely, if ever had seen join tables used in a professional setting, as they introduce a level of complexity that is very difficult to troubleshoot effectively.

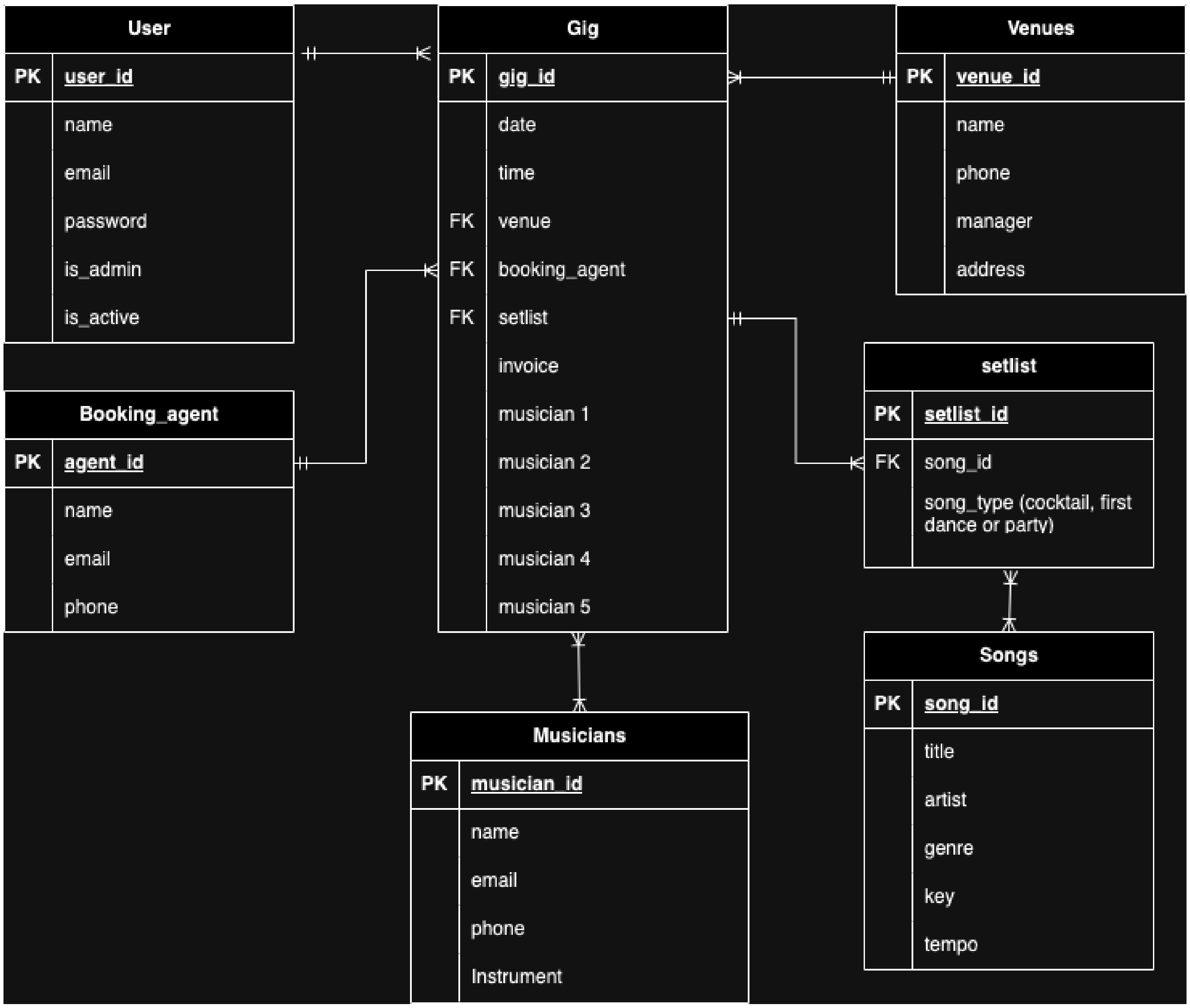
With the deadline I had to meet, it was highly unlikely I was going to be able to create a functioning many-to-many join table, so I changed my design.

......but not before trying to make a SECOND join table for the musicians....



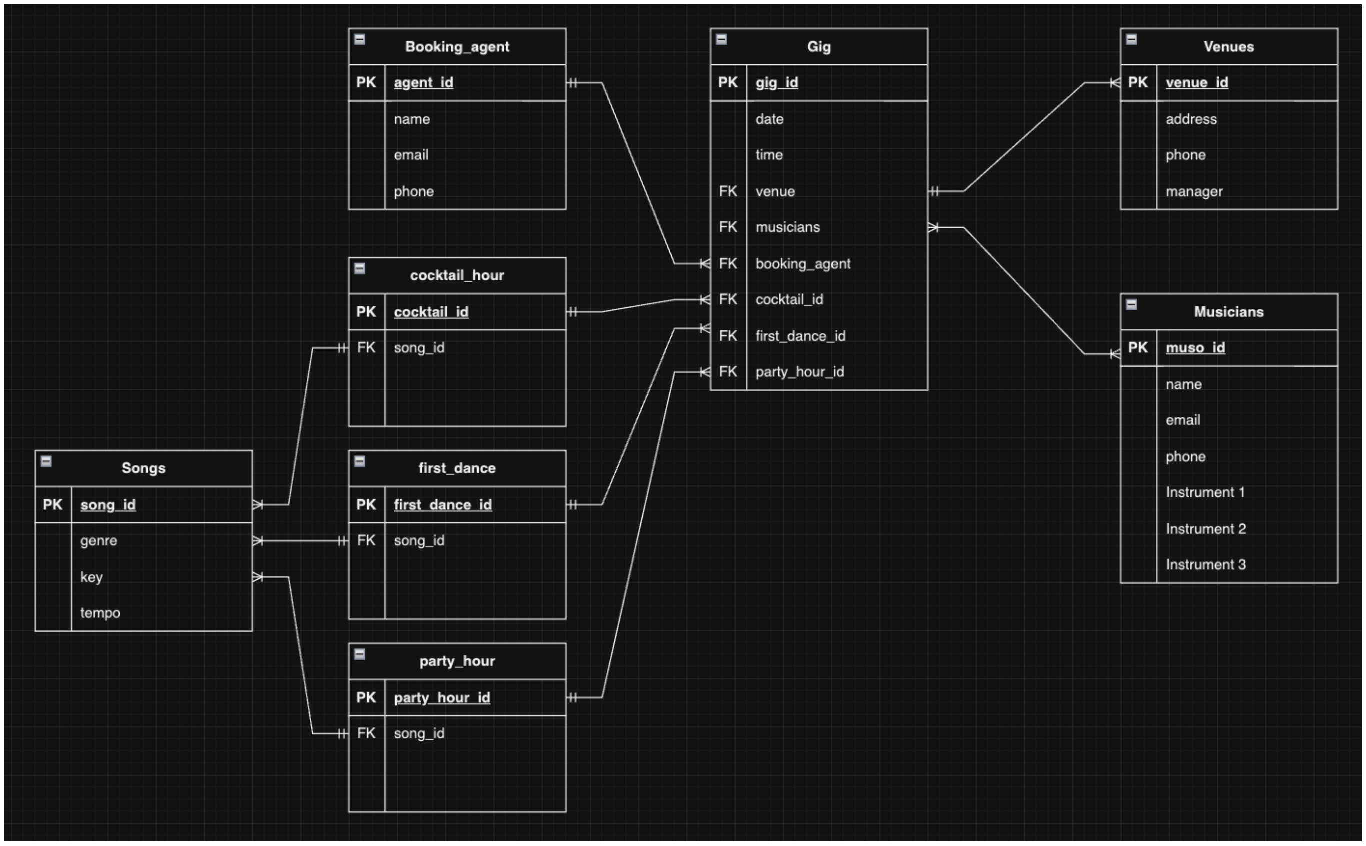
I decided that I needed this table as my first design was only going to allow me to assign a single musician to each `Gig` entry, which was not fit for purpose, as all each entry was going to have at least 2 musicians at it. My solution was to add a `Band` table. This was to account for the nature of the work. It's not uncommon for wedding bands to have a different group of musicians for any given gig.

I attempted to get around this by changing my design again, but to no avail....



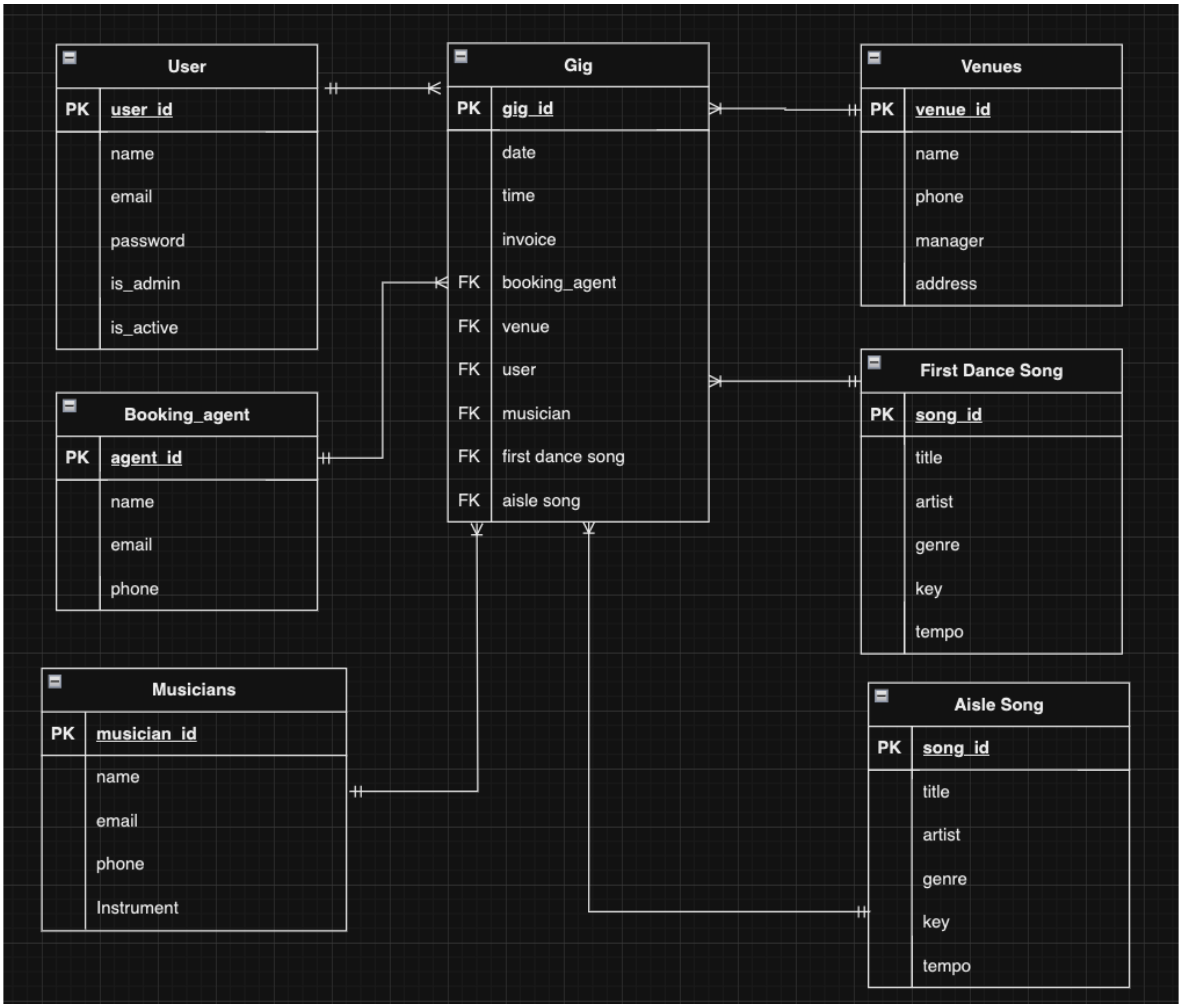
This was my attempt to get around the join table between musicians and gigs, but (of course) the same problems came up again and again.

At some point I also attempted to get around the join table for the `songs` and `setlist` table by creating 3 different tables. The problem I found here was that a `First Dance` song could be a `Cocktail Hour` etc. I had just created 3 versions of the same problem.



Finally, I settled on a much simpler design where it was just many different One-to-Many relationships. I wasn't happy about it, but I was going to meet the deadline.

I ended up with this:



No more join tables. I boiled the functionality down to just a few components. Every gig still had a `First Dance Song`, a `Aisle Song` and at least one `Musician` functioning as a "Band Leader".

In retrospect, I wish I had mastered the join table, or designed something completely different.

I think the main skill that I lacked for this task was my ability design an app that met the criteria and matched with my skillset in Flask, PSQL and Python. The initial design was way too complex. I fundamentally didn't understand the complexities of the Many-To-Many relationship in a relational database or who to correctly implement them. I bit off FAR more than I could chew.

To my credit, I recognised my error and course corrected. I didn't "throw good money after bad".

I think if I were doing this project again, I would design a much simpler app. I would do either just the `Musicians` or just the `Setlists` components of the apps on their own. Each of these problems would have met the scope of the requirements of the assignment.

I think I would've favoured the `Setlist` idea, as I would be able to take advantage of using the Spotify API, which allows you to populate the table with data about each song, specifically the `Title, Artist, Key, Genre, Tempo.` You can also take advantage of existing Wedding Playlists.

Once I had confidence in making a smaller idea work, I would've gained the knowledge, skills and confidence to expand and iterate on my initial idea by adding more features. I might have eventually gotten to a point where I was able to use the Join Tables that would've bought my initial design into reality.

Question 7

```
Explain control flow, using an example from the JavaScript programming
language
```

Explains the concept of control flow in programming

"Provides a thorough explanation of control flow in programming"

Control flow refers to the order that lines of codes are run by the computer. In the case of JavaScript, it's top to bottom. The code will run from the top to the bottom, unless there is a condition within the code that allows the code to stop running early. This can be done using Loops, Conditionals, and Functions.

Loops

Loops are pieces of code that iterate until there is nothing left to loop, or a condition is met. This means you have to be careful to not write code that will always be met, as you'll have an endless loop.

For

The first kind of loop to look at is a `for` loop, which will run until the conditions are no longer `True`.

```
let numbers = [1, 2, 3, 4, 5];
// Initialises an array of number.

let sum = 0;
// Initialises a variable of `sum`.

for (let i = 0; i < numbers.length; i++) {
/*
`for` means this will be a for loop, which will execute until the statement
following is no longer `True`.
`let i = 0` sets the value of index to zero. We're going to look at the
first element first.
`i < numbers.length;` is the condition that needs to be met. As long as the
value of `i` is less than the length of the `numbers` array (which is 5),
the loop will continue to iterate.
`i++` adds + 1 to the value of `i` every time the condition is met.

In this example, on the 6th ireration, i = 5, therefore, i < numbers.length
= false, and the loop breaks.
*/

    sum += numbers[i];
// `sum` will store the value of the `numbers` at a given interval. It will
add the value to the total value of `sum` after each iteration.
}

console.log("The sum of the numbers is: " + sum);
// This will print out the value the total value of the `numbers` array,
added together with each iteration and stored in the `sum` variable.
```

The above code would have this output:

```
The sum of the numbers is: 15
```

While

A `while` loop is quite similar to a for loop, but it's better to use it when you don't know how many times the loop is going to have to iterate.

```
const eliteFourBattleTeam = ['Venusaur', 'Charizard', 'Blastoise',
'Alakazam', 'Gengar', 'Gyarados'];

while (eliteFourBattleTeam.length > 0) {
/*
`while` establishes a `while` loop.
`eliteFourBattleTeam.length > 0` means that `while` the length of the array
is greater than zero, it will keep looping.
*/

        const fightingPokemon = eliteFourBattleTeam.shift();
// This is a variable that is created to hold the value of the pokemon from
the array that gets `shifted` by the `eliteFourBattleTeam.pop()` method.
Withouth this line, this would be an infinte loop, as the condition would
always be true.


        console.log(fightingPokemon + ", I choose you!"); }
```

```
// This will concatenate the name of the pokemon shifted with the phrase ",
I choose you!" and print it to the console.
```

The above code would have this output:

```
Venusaur, I choose you!
Charizard, I choose you!
Blastoise, I choose you!
Alakazam, I choose you!
Gengar, I choose you!
Gyarados, I choose you!
```

Do-while

A `do-while` loop is similar to a while loop, but the `do` portion guarantees that the loop runs at least once. This is because the `do` is outside of the `while` loop.

```
const eliteFourBattleTeam = ['Venusaur', 'Charizard', 'Blastoise',
'Alakazam', 'Gengar', 'Gyarados'];

let fightingPokemon;
// This is a variable that needs to be declared outside of the loop. This is
where we will send the pokemon that are being "sent out to fight".

do {
    fightingPokemon = eliteFourBattleTeam.shift();
// `shift` means that we're starting from the front of the array. Venusaur
goes our to fight first. The last one out is Gyarados.

    console.log(fightingPokemon + ", I choose you!");
// This will concatenate the name of the pokemon shifted with the phrase ",
I choose you!" and print it to the console.

} while (eliteFourBattleTeam.length > 0);

// This is the `while` loop. It works the same as the one above, we just had
to `do` the first iteration outside of the `while` loop.
```

The above code would have the same output as the `while` loop above:

```
Venusaur, I choose you!
Charizard, I choose you!
Blastoise, I choose you!
```

```
Alakazam, I choose you!
Gengar, I choose you!
Gyarados, I choose you!
```

Conditionals

Conditional statements are used in parts of the code where a condition needs to be satisfied to decide where to go next. Loops will use conditional logic, but these are not the same as loops, as a conditional statement don't have to be from a loop. The most common conditionals in JavaScript are `if`, `if else`, `else` and `switch`.

if

An if statement could be as simple as this:

```
let stopSign = true;
// invoke `stopSign` as a variable with a value of `true`.

if (stopSign) {
// pass `stopSign` as an argument in the `if` statement. If it is true
(which it is), we move to the next line of code.

        console.log(`There's a stop sign! Stop!`);
}
// `There's a stop sign! Stop!` gets printed to the console.
```

if else

`if else` statements give us a bit more control. We can have an option for what to do if the `if` statement is not satisfied:

```
let stopSign = true;

if (stopSign) {
    console.log(`There's a stop sign! Stop!`);
// all the same as above to this point.

} else {
    console.log(`There's no stop sign. You can proceed.`);
// the `else` statement gives us an option if the `stopSign` were to become
false.
}
```

To make it a little more complex, we can add some loops.

This also incorporates a function, which is explained below:

```
let nrlTeams = ['Roosters', 'Raiders', 'Storm', 'Rabbitohs', 'Eels',
'Panthers', 'Sharks', 'Sea Eagles', 'Knights', 'Dolphins'];
// This is an array of mascots of 10 of the teams in the National Rugby
League.


function isTop4(team) {
// this is a fucntion that establishes isTop4 as a method and passes `team`
as an argument.
    return ['Roosters', 'Knights', 'Storm', 'Rabbitohs'].includes(team);
// This puts the above teams into the `isTop4` method.
}


for (let i = 0; i < nrlTeams.length; i++) {
/*
`for` is a for loop that will contiue while the conditons are true.
`let i = 0` starts at the first element of the array.
`i < nrlTeams.length` means the loop will continue for the length of the
nrlTeams array, which is 10.
`i++` increases the value of `i` by 1 for each iteration.
*/

    let team = nrlTeams[i];
// this calls the `team` variable we passed as an argument before and states
that it's equal the element at each index of nrlTeams.


    if (isTop4(team)) {
// if the teams being iterated over also appear in the `isTop4` fucntion,
this statement will be printed to the console.
        console.log(`${team} is in the top 4. Go the ${team}!`);
// if the team iterated over does not also appear in the `isTop4` function,
this else statement will be intiated and the console log will print.
    } else {
        console.log(`${team} is not in the top 4.`);
    }
}
```

The output of this code would be:

```
Roosters is in the top 4. Go the Roosters!
Raiders is not in the top 4.
Storm is in the top 4. Go the Storm!
Rabbitohs is in the top 4. Go the Rabbitohs!
Eels is not in the top 4.
Panthers is not in the top 4.
Sharks is not in the top 4.
Sea Eagles is not in the top 4.
Knights is in the top 4. Go the Knights!
Dolphins is not in the top 4.
```

Switch

`Switch` statements are a bit like `if` statements. The difference is that there is multiple return statements that match a single condition. Once the condition is satisfied, the code stops running.

```
function getTeamOfTheWeek(num) {
// intiates the function and passes num as a argument

        switch (num) {
                case 1:
                        return 'Roosters';
                case 2:
                        return 'Raiders';
                case 3:
                        return 'Storm';
                case 4:
                        return 'Knights';
                case 5:
                        return 'Rabbitohs';
                case 6:
                        return 'Eels';
                case 7:
                        return 'Panthers';
                case 8:
                        return 'Sharks';
                case 9:
                        return 'Sea Eagles';
                case 10:
                        return 'Titans';
// 10 different options for the `switch` statement that all correspond to a
number and have their own return statements.
                default:
```

```
                          return 'Unknown Team';
// this is what would be returned if no number was passed
        }
}

console.log(getTeamOfTheWeek(4));

// 'Knights' is the obvious correct answer.....
```

Functions

`Functions` can also be a used as a part of more complex loops.

```
function isTop4(team) {
    return ['Roosters', 'Knights', 'Storm', 'Rabbitohs'].includes(team);
}
/*
This function is intiated as `isTop4`, and passes `team` as an argument.
The return statement includes an unnamed array with the 4 teams that we want
to be identified as Top 4 sides.

We use the .includes() method, which is part of JavaScript. It checks if the
value of the team being iterated over in the original nrlTeams array matches
any of those included in the `isTop4` function.
*/
```

We could add extra conditions to this function as well, if your need required it:

```
function isTop4(team) {
    const topTeams = ['Roosters', 'Knights', 'Storm', 'Rabbitohs'];
// `topTeams` is now an array.

    return topTeams.includes(team) && typeof team === 'string';
// This now checks whether or not the teams in the origianl array also have
the `typeof` `string`. They're obvisouly all strings, but this is just an
example of adding extra conditons to a function.
}
```

References

Cleary, Rianna, Control Flow in JavaScript, 2020,
https://medium.com/@rianna.cleary/control-flow-in-javascript-9c63d0c98bb9

Geeksforgeeks.org, Conditional Statements in JavaScript, 2024
https://www.geeksforgeeks.org/conditional-statements-in-javascript/?ref=ml_lbp

Question 8

```
Explain type coercion, using examples from the JavaScript programming
language
```

Explains the concept of type coercion in programming
"Provides a thorough explanation of type coercion in programming"

Type Coercion

Type coercion is the implicit conversion from one data type to another one. This could be from a number to a string, a string to a boolean, a boolean to a number etc.

This is not quite the same as Type Conversion, as that is a change that happens either explicitly or implicitly. Type Coercion only happens implicitly, meaning that JavaScript automatically determines from the context of the code what should happen to the datatype.

Number to String

```
let num = 42;
let lifeTheUniverseAndEverything = "The Answer to the Ultimate Question of
Life, the Universe, and Everything is " + num;
console.log(lifeTheUniverseAndEverything);
```

In this instance, the `num` of `42` has been coerced from a number to a string, and concatenated with the string above to output"

```
The Answer to the Ultimate Question of Life, the Universe, and Everything is
42.
```

String to Number

The coercion of string to number is easiest to demonstrate with some simple arithmetic.

```
let x = 100 + + `99`;
// `99` is treated as a string, because it's in back ticks. If there was
```

```
only one plus sign, it would coerce `100` to a string and concatenate with
99 to print out 10099.
// As we've used the `+ +`, it instead coerces `99` to be a number and
prints 199.
let y = 12 * `12`;
// `12` is coerced to be a number and therefore multiplies 12 * 12.

let z = 10 % `5`;
// `5` is coerced to be a divider. As 10/5 is an even number, the answer
will be 0 becuase there is no remainder.

console.log(x);
console.log(y);
console.log(z);

// The output will be:
199
144
0
```

Boolean to Number

Boolean's can be converted to numbers quite easily, as both possible values have a number equivalent. True is considered to equal 1, and false is considered to equal 0.

```
let a = 42 + true;
// this is the same as saying 42 + 1

let b = 42 + false;
// this is the same as saying 42 + 0

console.log(a);
// this will output 43

console.log(b);
// this will output 42
```

Equality Operator

If you use the strict equality operator `===`, you won't get any type coercion. However, if you use the loose equality operator `==`, you will get type coercion, depending on the context.

This is easiest specified with boolean values.

```
let x = (42 == `42`)
let y = (42 === `42`)
let z = (1 == true)

console.log(x)
// Output true because `42` was coerced to a number.

console.log(y)
// Output false because types differ, and the `42` was not coerced.

console.log(z)
// Output true becuase 1 = true.
```

However:

```
let x = (true == `true`)

console.log(x)
// This will not print out true. This will print out NaN, meaning `Not a
Number`. This is because the first `true` in this instance is coerced into a
number, 1 to be specific. Therefore the second `true` is taken as a string,
and the type cannot be coerced to a number.
```

References

Jha, Atul, Coercion & Type Conversion, 2023, https://medium.com/@atuljha2402/understanding-javascript-type-coercion-type-conversion-a2ce84c00331

geeksforgeeks.org, What is Type Coercion in JavaScript?, 2024, https://www.geeksforgeeks.org/what-is-type-coercion-in-javascript/

Question 9

```
Explain data types, using examples from the JavaScript programming language
```

Explains the concept of data types in programming
"Provides a thorough explanation of data types in programming"

Like all programming languages, JavaScript has it's own built in data structures. They might not be exactly the same across all languages. Python differentiates between a floating-point number or `float` and an integer or `int`, where as JavaScript has a simpler number system and just uses `Numbers` and `BigInt` when they reach the extremities of how large a number JS can handle. But they both use a numeric data type of some sort. They also use strings, booleans and objets.

Data Types

JavaScript has 8 datatypes that it works with:

- Number
- String
- Boolean
- Null
- Undefined
- Symbol
- BigInt
- Object

Each of these data types (except for `null`) can have their type checked with the `typeof` operator. Null requires the user to test with `=== null` .

All except for the `Object` datatype are known as Primitive datatypes.

Primitive datatypes are immutable values at their most basic level. They don't reference anything else within an object, and once created they can't be changed.

Number

In JavaScript, `numbers` represent both integers and floating-point numbers

```
let num = 42;
let pi = 3.14159265359
```

String

A `string` is simply a sequence of characters used to represent text. Strings are immutable, and usually have inverted commas around them. Single '' or double "" both work, as do backticks "``"

```
let name = "Herschel Krustofsky";
let greeting = `Hey Hey!`

```

Boolean

A `boolean` data type can have one of exactly two values; `true` or `false`.
With a logical NOT operator of `!` these values can be inverted, but there are still only two options.

```
let isTrue = true;
// this is true, and has a numeric value of 1 when coerced.

let isFalse = false;
// this is false, and has a numeric value of 0 when coerced.

let isNotTrue = !true;
// this is false, and has a numeric value of 0 when coerced.

let isNotFalse = !false;
// this is true, and has a numeric value of 1 when coerced.
```

Null

Null can have exactly one value; `null`. This is the intentional absence of an object value.

```
let emptyValue = null;
```

Undefined

Undefined is similar to `null`, but this more about the value of a variable that has been declared, but hasn't been assigned a value yet.

```
let undefinedVar;
```

Symbol

`Symbol` is a newer data type in JavaScript. They create a completely unique identifier, even if they look the same. They can be used for hiding information. They're immutable, so they cannot be changed once created.

```
const sym1 = Symbol(`cat`);
const sym2 = Symbol(`cat`);

console.log(sym1 === sym2);

// This would output as false. Even though we've passed `cat` as an argument
for both instances of a Symbol, they're not equal. This is because of the
unique identifier that isn't displayed.
```

BigInt

`BigInt` is used to represent whole numbers that are larger than 2^53 -1. this is the highest number that JavaScript can reliable represent within the `Number` data type. Expressed as an integer, the value is `9007199254740991`. It can be used by placing an `n` at the end of a very large integer.

```
let bigNum = 1234567890123456789012345678901234567890n;
```

Object

An `Object` in JavaScript is the way that we store data. They come in the form of key-value pairs and are instantiated using curly braces {}. The use for objects is limited only by your imagination. Let's say you wanted to store a biography of the singer Seal that could be accessed for an API.

```
let seal = {
    firstName: "Seal",
    lastName: "Henry Olusegun Olumide Adeola Samuel",
    birthDate: "1963-02-19",
    nationality: "British",
// These are simple key value pairs seperated by a colon :
    genres: ["Soul", "R&B", "Pop"],
    famousSongs: ["Kiss from a Rose", "Crazy", "Love's Divine"],
// These key value pairs use an array.
    awards: {
        grammys: 4,
        britAwards: 3,
        other: ["MTV Video Music Award", "Ivor Novello Award"]
    },
// This is a nested object. Key value pairs for the awards he's won multiple
times, and an array for awards he's only won once.
    activeYears: "1987-present",

    biography: function() {
// This is a function as part of a key value pair.
        return `${this.firstName} is a British singer known for hits like
"${this.famousSongs[0]}" and "${this.famousSongs[1]}". He has won
${this.awards.grammys} Grammy Awards and ${this.awards.britAwards} Brit
Awards.`;
// This function calls various pieces of data from the above key value pairs
of this object to create a print statement which sumerises Seal's
achievements.
        }
};
console.log(seal.biography());
// This is how we print the function to the console.
```

The output of this function would be:

```
"Seal is a British singer known for hits like "Kiss from a Rose" and
"Crazy". He has won 4 Grammy Awards and 3 Brit Awards."
```

References

MDN Web Docs, JavaScript data types and data structures, 2024.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

W3Schools, JavaScript Data Types, 2024,
https://www.w3schools.com/js/js_datatypes.asp

geeksforgeeks.org, JavaScript data Types, 2023,
https://www.geeksforgeeks.org/javascript-data-types/

Question 10

```
Explain how arrays can be manipulated in JavaScript, using examples from the
JavaScript programming language
```

"Demonstrates an extensive ability to manipulate arrays"

An array is variable that acts a bit like a container to store data of various types that an later be assess with another variable. You can have multiple datatypes stored at once.

There are many different ways that arrays can be manipulated in JavaScript.

Destructive Manipulations

Destructive manipulation methods allow you to change the array in many different ways, but the changes are permanent.

To declare an array, you just have to use `let`:

```
let MyArray = [`hello`, 1, true, 1.2];
```

.push()

Push allows you to append elements to the end of an array.

```
let pokemon = [`Bulbasaur`, `Charmander`, `Squirtle`];
pokemon.push(`Pikachu`, `Pidgey`);
console.log(pokemon)
// [`Bulbasaur`, `Charmander`, `Squirtle`,`Pikachu`, `Pidgey`]
```

.unshift()

Unshift works the same as Push, but it adds the elements to the front of the array.

```
let pokemon = [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`, `Pidgey`];
pokemon.unshift(`Onyx`, `Spearow`);
console.log(pokemon)
```

```
// [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`,
`Pidgey`]
```

.pop()

Pop removes elements to the end of the array.

```
let pokemon = [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`,
`Pikachu`, `Pidgey`];
pokemon.pop();
console.log(pokemon)
// [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`]
```

.shift()

Shift removes elements to the front of the array.

```
let pokemon = [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`,
`Pikachu`];
pokemon.shift();
console.log(pokemon)
// [`Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`]
```

.splice()

Splice can be used to both remove and add elements from and array. You have to pass and index as an argument.

If I wanted to remove everything after index 3 in my array, it would look like this:

```
let pokemon = [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`,
`Pikachu`];
pokemon.splice(3);
// => [`Squirtle`, `Pikachu`]

console.log(pokemon)
// [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`]
```

But if I wanted to remove a range of elements from the array, it would look like that:

```
let pokemon = [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`,
`Pikachu`];
pokemon.splice(2,4);
```

```
// => [`Bulbasaur`, `Charmander`, `Squirtle`]

console.log(pokemon)
// [`Onyx`, `Spearow`, `Pikachu`]
```

If I wanted to add elements into that array in the position where I've just removed elements from, the code might look like this:

```
let pokemon = [`Onyx`, `Spearow`, `Bulbasaur`, `Charmander`, `Squirtle`,
`Pikachu`];
pokemon.splice(2, 4, `Ekans`, `Koffing`);
// => [`Bulbasaur`, `Charmander`, `Squirtle`]

console.log(pokemon)
// [`Onyx`, `Spearow`, `Ekans`, `Koffing`, `Pikachu`]
```

concat()

Concat can also be used by passing an argument to add an element to the end of the array:

```
let popularPokemon = [`Pikachu`, `Clefairy`, `Eevee`];
popularPokemon = popularPokemon.concat(`Charmander`);
console.log(popularPokemon);
// [`Pikachu`, `Clefairy`, `Eevee`, `Charmander`]
```

Non-Destructive Manipulations

Non-Destructive methods of manipulating arrays allow you to make copies of the original array that can be changed while still keeping the original array intact, and therefore maintaining in the integrity of the original data.

Instead of using `let`, you can use `const` which is short for `constant`, as the original array won't be changing.

```
const pokemon = [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`,
`Geodude`];
```

.slice()

Slice will copy part of an array that is parsed, and return a copy of those elements in a new array. The first index parsed is inclusive, but the second is not.

If I wanted to remove the middle 3 elements of this array as a "slice" I would have to say `1` to catch `Charmander`, but 4 to make it stop at `Pikachu`.

This will create a seperate array, while leaving the original intact.

```
const pokemon = [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`,
`Geodude`];
const slicedPokemon = pokemon.slice(1,4);
console.log(slicedPokemon);
// [`Charmander`, `Squirtle`, `Pikachu`]
console.log(Pokemon);
// [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`, `Geodude`]
```

If we used slice with only 1 argument passed, it would take slice from the index specified till the end of the array.

```
const pokemon = [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`,
`Geodude`]; const slicedPokemon = pokemon.slice(2);
console.log(slicedPokemon);
// [`Squirtle`, `Pikachu`, `Geodude`]
console.log(Pokemon);
// [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`, `Geodude`]
```

.filter()

Filter allows us to create a condition to only put certain elements of the array into the new array.

If I this array:

```
const pokemon = [`Charizard`, `Pikachu`, `Cloyster`, `Bulbasaur`, `Jolteon`,
`Cubone`];
```

And I wanted a seperate array that only had Pokemon that start with the letter C, the code might look like this:

```
const pokemon = [`Charizard`, `Pikachu`, `Cloyster`, `Bulbasaur`, `Jolteon`,
`Cubone`];

function pokemonThatStartWithC(pokemonName) {
        return pokemonName[0] === `C`
}
const pokemonStartingWithC = pokemon.filter(pokemonThatStartWithC);
console.log(pokemonStartingWithC);
```

```
// [`Charizard`, `Cloyster`, `Cubone`]
console.log(pokemon);
// [`Charizard`, `Pikachu`, `Cloyster`, `Bulbasaur`, `Jolteon`, `Cubone`]
```

However, with this code we're only checking for a capital `C`. We should also check for a for a lowercase `c`.

```
const pokemon = [`Charizard`, `Pikachu`, `Cloyster`, `Bulbasaur`, `Jolteon`,
`Cubone`];

function pokemonThatStartWithC(pokemonName) {
        return pokemonName.charAt(0).toLowerCase() === `c`;
/*
`charAt(0)` focuses on the character at the zero index of the element.
`.toLowerCase()` will change the first character of the element to
lowercase, if it is not already lowercase.
`=== `c`;` checks if the now lowercase first character is equal to `c`.
*/
}
const pokemonStartingWithC = pokemon.filter(pokemonThatStartWithC);
console.log(pokemonStartingWithC);
// [`Charizard`, `Cloyster`, `Cubone`]
console.log(pokemon);
// [`Charizard`, `Pikachu`, `Cloyster`, `Bulbasaur`, `Jolteon`, `Cubone`]
```

.concat() and spread ...

The non-destructive way to use concat is to make a third array to simply concatenate two arrays together.

```
let starterPokemon = [`Bulbasaur`, `Charmander`, `Squirtle`];
let popularPokemon = [`Pikachu`, `Clefairy`, `Eevee`];
let myPokemon = starterPokemon.concat(popularPokemon);
console.log(myPokemon);
// [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`, `Clefairy`, `Eevee`]
```

Another way to concatenate is to use the spread operator, which is just three dots `...`

```
const starterPokemon = [`Bulbasaur`, `Charmander`, `Squirtle`];
const startersAndPikachu = [...starterPokemon, `Pikachu`];
console.log(startersAndPikachu);
// [`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`]
```

```
const startersPikachuMagikarp = [`Magikarp`, ...startersAndPikachu];
console.log(startersPikachuMagikarp);
// [`Magikarp`,`Bulbasaur`, `Charmander`, `Squirtle`, `Pikachu`]
```

.map()

Map lets you create a new array from the original, while also performing some kind of transformation. Let's say I wanted to make the names of all these Pokemon start with a lowercase letter:

```
const pokemon = [`Charizard`, `Pikachu`, `Cloyster`, `Bulbasaur`, `Jolteon`,
`Cubone`];

const transformedPokemon = pokemon.map(pokemonName => {
    return pokemonName.toLowerCase();
});
console.log(transformedPokemon);
// [ 'charizard', 'pikachu', 'cloyster', 'bulbasaur', 'jolteon', 'cubone']
```

References

Cleary, Rhianna, Array Manipulation in JavaScript, 2020, https://medium.com/@rianna.cleary/array-manipulation-in-javascript-233f8f13a245

Ayodeji, Bolaji, How to Manipulate Arrays in JavaScript, 2019, https://www.freecodecamp.org/news/manipulating-arrays-in-javascript/

Question 11

```
Explain how objects can be manipulated in JavaScript, using examples from
the JavaScript programming language
```

Demonstrates an ability to manipulate objects

"Demonstrates an extensive ability to manipulate objects"

Objects are the data structure that uses key-value pairs in JavaScript. They can be manipulated in many ways, including creating, accessing, modifying, and deleting properties.

Creating Objects

Objects can be created in two ways. You can either use the `new Object()` constructor, or just literal notation between curly braces {}

```
const person = new Object();
person.firstname = "Bruce";
person.surname = "Wayne";
person.age = "40";
person.city = "Gotham City";
person.occupation = "Batman";

// This is using the constructor

const person = {
        firstname: "Bruce",
        surname: "Wayne",
        age: "40",
        city: "Gotham City",
        occupation: "Batman"
}
// This is the same object, just created with literal notation.

const car = {
        make: "Wayne Enterprises",
        model: "Batmobile",
        year: "2008"
};
// This is another example of an object created with literal notation.
```

Accessing Properties

Now that we've got a couple of objects, we can access the information held in them using dot notation and (`.`) or bracket notation (`[]`)

```
const person = {
        firstname: "Bruce",
        surname: "Wayne",
        age: "40",
        city: "Gotham City",
        occupation: "Batman"
}

const car = {
        make: "Wayne Enterprises",
        model: "Batmobile",
        year: "2008"
};

console.log(person.firstname);
// Output: Bruce
console.log(car["model"]);
// Output: Batmobile
```

Modifying Properties

You can modify the properties of the objects by simply stating what you want the new value to be:

```
const person = {
        firstname: "Bruce",
        surname: "Wayne",
        age: "40",
        city: "Gotham City",
        occupation: "Batman"
}

const car = {
        make: "Wayne Enterprises",
        model: "Batmobile",
        year: "2008"
};

person.age = 41;
car["year"] = 2024;

console.log(person.age);
// Output: 41
console.log(car["year"]);
// Output: 2024
```

Adding Properties

Adding properties is also quite straightforward:

```
const person = {
        firstname: "Bruce",
        surname: "Wayne",
        age: "41",
        city: "Gotham City",
        occupation: "Batman"
}

const car = {
        make: "Wayne Enterprises",
        model: "Batmobile",
        year: "2024"
};
person.height = 185;
car.colour = "Black";

console.log(person);
/*
Output:
{
  firstname: 'Bruce',
  surname: 'Wayne',
  age: '41',
  city: 'Gotham City',
  occupation: 'Batman',
  height: 185
}
*/

console.log(car);
```

```
/*
Output:
{
  make: 'Wayne Enterprises',
  model: 'Batmobile',
  year: '2024',
  colour: 'Black'
}

NB: These are not JSON format. They're stings made to look like JSON format.
*/
```

Deleting Properties

Deleting properties is also very straightforward once you know how to correctly access elements of the object:

```
const person = {
        firstname: "Bruce",
        surname: "Wayne",
        age: "41",
        city: "Gotham City",
        occupation: "Batman",
        height: 185
}

const car = {
        make: "Wayne Enterprises",
        model: "Batmobile",
        year: "2024",
        colour: "Black"
};

delete person.occupation;
car.make = "Lamborghini";
car.colour = "Yellow";
delete car.model;
// Gotta keep his identity a secret!

console.log(person);
/*
Output:
{
  firstname: 'Bruce',
  surname: 'Wayne',
  age: '41',
  city: 'Gotham City',
  height: 185
}
*/

console.log(car);
/*
Output:
{
  make: 'Lamborghini',
  year: '2024',
  colour: 'Yellow'
}

*/
```

Iterating Over Properties

Iterating over properties allows you to create a `for` loop that could be used to transform the data in some way.

```
const person = {
        firstname: "Bruce",
        surname: "Wayne",
        age: "41",
        city: "Gotham City",
        occupation: "Batman",
        height: 185
};


for (let key in person) {
// This starts a for loop, establishes key as a variable, and tells it to
check in the instance of `person`

        if (typeof person[key] === 'string') {
// An `if` statement was required as the height key and value pair is a
number and not a string.

                person[key] = person[key].toUpperCase();
// This will turn value at each key that is a string to all uppercase
lettering.
        }
}

console.log(person);
/*
Output:
{
        "firstname": "BRUCE",
        "surname": "WAYNE",
        "age": "41",
        "city": "GOTHAM CITY",
        "occupation": "BATMAN",
        "height": 185
}
*/
```

Checking Property Existence

You can check if a particular property is present in an object with the `hasOwnProperty()` method. You could also use the `in` operator. But if we attempt to access a non-existent property it will return `undefined`.

```
const person = {
    firstname: "Bruce",
    surname: "Wayne",
    age: "41",
    city: "Gotham City",
    occupation: "Batman",
    height: 185
};

console.log(person.hasOwnProperty("firstname"));
// Output: true
console.log("age" in person);
// Output: true
console.log(person.hasOwnProperty("address"));
// Output: There is no address property, so `hasOwnProperty()` returns false
console.log(person.address);
// Output:  we didn't use `hasOwnProperty()` or `in`, and it doesn't exist,
so it returns undefined
```

Adding a Function

Functions can also be added as part of a property of an object. They can access different properties of the object.

```
const person = {
        firstname: "Bruce",
        surname: "Wayne",
        age: 41,
        city: "Gotham City",
        occupation: "Batman",
        height: 185,
        greet: function() {
// greet has been added as a property of the object and we've nested a
fucntion that will print out a greeting from Batman.
                console.log(`I'm ${this.occupation}! The Vengence of
${this.city}!`);
// This will print the greeting to the console. It uses the properties of
`occupation` and `city`. If these properties were to change, the output of
this function would change with them.
        }
};

person.greet();
// Output: I'm Batman! The Vengence of Gotham City!
// We don't need to say console.log, as that is already apart of the
function.
```

References

MDN Web Docs, JavaScript object basics, 2024,
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Basics

Talha, Md Abu, JavaScript Object Manipulation, 2018,
https://medium.com/infancyit/javascript-object-manipulation-5d1145cf06ef

geeksforgeeks.org, Objects in JavaScript, 2024,
https://www.geeksforgeeks.org/objects-in-javascript/

Question 12

```
Explain how JSON can be manipulated in JavaScript, using examples from the
JavaScript programming language
```

"Demonstrates an extensive ability to manipulate JSON"

A `JSON` (JavaScript Object Notation) looks very similar to an object in JavaScript. The primary difference is that JSON is used mostly for data interchange in APIs and for configuration settings. An object, on the other hand, is a programming construct that can be found in multiple programming languages and is more likely to contain functions.

While it's technically possible to include functions in a JSON, it is uncommon because JSON is primarily used to store data in key-value pairs.

Parsing JSON

Parsing is the process of analysing a JSON string and creating an object that can be manipulated and accessed through your code. When you receive JSON strings from an API, they need to be parsed so they can be a useable JSON object.

```
const jsonString = '{"name": "Tony Stark", "alias": "Iron Man", "age": 45}';
const jsonObject = JSON.parse(jsonString);


console.log(jsonObject.name);
// Output: Tony Stark
console.log(jsonObject.alias);
// Output: Iron Man
console.log(jsonObject.age);
// Output: 45
```

Creating JSON

The inverse of parsing is creating a JSON string from a JavaScript object. This is often done when you want to send data to a web server or store it in a format that can be easily transmitted and stored.

```
const person = {name: "Tony Stark", alias: "Iron Man", age: 45};
const jsonString = JSON.stringify(person);

console.log(jsonString);
// Output:
// {"name": "Tony Stark", "alias": "Iron Man", "age": 45}
```

Accessing JSON

Once parsed, the accessing of a JSON is the same way you would access the data and properties from any other JavaScript Object.

```
const jsonObject = {name: "Tony Stark", alias: "Iron Man", age: 45};

console.log(jsonObject.name);
// Output: Tony Stark
console.log(jsonObject.alias);
// Output: Iron Man
console.log(jsonObject.age);
// Output: 45
```

Modifying JSON

Once you've modify key-value pairs in a JSON Object you should convert it back to a JSON string:

```
let jsonObject = {name: "Tony Stark", alias: "Iron Man", age: 45};

jsonObject.age = 46;

const jsonString = JSON.stringify(jsonObject);

console.log(jsonString);
// Output:
// {"name": "Tony Stark", "alias": "Iron Man", "age": 46}
```

Adding and Removing JSON Properties

Adding and Removing to and from a JSON object is very straightforward and the same as any other object.

```
let jsonObject = {name: "Tony Stark", alias: "Iron Man", age: 46};

jsonObject.aiAssistant = "J.A.R.V.I.S";
// J.A.R.V.I.S has been added as Tony's AI assistant.

console.log(jsonObject.aiAssistant);
// Output: J.A.R.V.I.S

delete jsonObject.aiAssistant;
// J.A.R.V.I.S has become Vision, so Tony no longer has an AI assistant.

jsonObject.aiAssistant = "F.R.I.D.A.Y";
// After much searching, Tony found a replacement for J.A.R.V.I.S in
F.R.I.D.A.Y

console.log(jsonObject.aiAssistant);
// Output: F.R.I.D.A.Y

const jsonString = JSON.stringify(jsonObject);

console.log(jsonString);
// Output:
// {"name": "Tony Stark", "alias": "Iron Man", "age": 46, "aiAssistant":
"F.R.I.D.A.Y"}
```

Nested JSON

Nesting allows you to add arrays as part of as part of the key-value pairs.

```
let jsonString = `{
        "name": "Tony Stark",
        "alias": "Iron Man",
        "age": 45,
        "aiAssistant": "F.R.I.D.A.Y",
        "abilities": [
                "Genius-level intellect",
                "Engineering prowess",
                "Combat skills"
        ],
        "equipment": [
                "Powered armour suit"
        ]
}`;
// As this is a long string, I've spaced it out to improve readability.
// Square brackets allow you to add arrays as the value.
```

```
let jsonObject = JSON.parse(jsonString);

console.log(jsonObject);
```

The output of this in a nicely formatted JSON would look like this:

```
{
        name: "Tony Stark",
        alias: "Iron Man",
        age: 45,
        aiAssistant: "F.R.I.D.A.Y",
        abilities: [
                "Genius-level intellect",
                "Engineering prowess",
                "Combat skills"
        ],
        equipment: [
        "Powered armour suit"
        ]
}
```

References

geeksforgeeks.org, How to create and Manipulating JSON data in JavaScript, 2024, https://www.geeksforgeeks.org/how-to-create-and-manipulatinag-json-data-in-javascript/

MDN Web Docs, Working with JSON, 2024, https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

W3schools, JavaScript JSON, 2024
https://www.w3schools.com/js/js_json.asp

Question 13

```
For the code snippet provided below, write comments for each line of code to
explain its functionality. In your comments you must demonstrates your
ability to recognise and identify functions, ranges and classes
```

```
class Car {
        constructor(brand) {
                this.carname = brand;
        }
        present() {
                return 'I have a ' + this.carname;
        }
}

class Model extends Car {
        constructor(brand, mod) {
                super(brand);
                this.model = mod;
        }
        show() {
                return this.present() + ', it was made in ' + this.model;
        }
}

let makes = ["Ford", "Holden", "Toyota"]
let models = Array.from(new Array(40), (x,i) => i + 1980)

function randomIntFromInterval(min,max) { // min and max included
        return Math.floor(Math.random()*(max-min+1)+min);
}

for (model of models) {

        make = makes[randomIntFromInterval(0,makes.length-1)]
        model = models[randomIntFromInterval(0,makes.length-1)]

        mycar = new Model(make, model);
        console.log(mycar.show())
}
```

"Demonstrates an extensive ability to recognise functions, ranges and classes"

This code snippet outputs sentences of text stating that it has a car of a particular `make` (brand) and `model` (year it was built).

Am example of the output would be:
"I have a Ford, it was made in 1995"

The `brand` is limited to the answers of Ford, Holden and Toyota, and the `model` is limited to years from 1980 to 2019.

Lets break it down and add comments!

This first section is the class of Car.

```
class Car {
// This declares a class of Car
        constructor(brand) {
// This invokes a constructer and passes brand as an argument
                this.carname = brand;
/*
"this" refers to the instance of the object being created.
"carname" is to a property of the object being created.
"brand" is the parameter being passed by the constructor.

This line of code tells us that in this instance of the object Car, create
the property of "carname", and make it equal to "brand".
*/
        }
        present() {
// present() is a method to contain the return statement.
                return 'I have a ' + this.carname;
/* This gives us the format of how we are going to present the object in
this class. In this instance, we're concatinating it into a string starting
with "I have a " and the "carname", which is going to be determined by a
function later in the code. */
        }
}
```

The next section is another class called Model, which extends from the Car class above.

```
class Model extends Car {
// This declares that we've created another class called Model, and that
it's going to inherit some information from the Car class above.
        constructor(brand, mod) {
// Creating another constructor, this time passing brand and mod as
arguments.
                super(brand);
// Super lets us call "brand" from the Car class. This means that brand will
have the same result as what is passed in the Car class.
                this.model = mod;
/*
"this" refers to the instance of the object being created.
"model" is to a property of the object being created.
"mod" is the parameter being passed by the constructor. This will store the
value generated when the `Model` class is intiated.
*/
        }
        show() {
// show() is a method to contain the return statement.
                return this.present() + ', it was made in ' + this.model;
/*
"return" displays whatever comes after it.
"this.present()" calls the result of what we got on the for "present()" at
the end of the Car class. `+` concatinates the result of "this.present" with
`, it was made in `. `+` again to concatinate with "this.model", which will
be equal to what we determine to be `mod` later in the function.
*/
        }
}
```

Next we have two arrays

```
let makes = ["Ford", "Holden", "Toyota"]
// This is the `makes` array with 3 objects, which in this case are famous
car makes.

let models = Array.from(new Array(40), (x,i) => i + 1980)
/* This is the `models` array.
`Array.from(new Array(40)` creates an array with 40 positions (index 0 to
39), each with a value of `undefined`.
`(x,i) => + 1980)` is an arrow function with two arguments; x and i. x
represents the element, and i represents the index of that element.
`i + 1980` means that x will be declared as 1980 + the value at the current
index. Therefore, the value at index 0 will be 1980 + 0 = 1980. The value at
index 1 will be 1980 + 1 = 1981, index 2 will be 1980 + 2 = 1982 etc, all
```

```
the way up to index 39 which will 1980 + 39 = 2019.
*/
```

Now for the function:

```
function randomIntFromInterval(min,max) { // min and max included
// Here we've declared a function, called it `randomIntFromInterval` and
passed `min` and `max` as arguments.
        return Math.floor(Math.random()*(max-min+1)+min);
/*
`return` means we're about to see something printed....
`Math.floor(Math.random()` generates a float of either 0 or 1.
`*(max-min+1)+min)` multiplies the random number generated by the value of
`max - min+1`. It has to be plus 1 because we're dealing with index values,
and JavaScript uses a zero index. It then adds the value of `min`. This
means that the value of `min` is included. While min is zero at the moment,
the value of min might change in the future, and this would need to be
debugged without `+min`.
At this point in the program, neither `max` or `min` have been declared
explicitly, but they will be implied in the next section.
*/
}
```

Finally, into the for loop:

```

for (model of models) {
// This is a for...of loop, which will iterate over an object such as a
string, array, map, set etc.
// `model` is declared as the variable that represents the element at each
itteration of the loop, which in this instance will be one for each year
specified within the range in the `models` array above.

        make = makes[randomIntFromInterval(0,makes.length-1)]
/*
`make` is declared as a variable.
`makes` is the array declared above containing ["Ford", "Holden", "Toyota"]
`randomIntFromInterval` generates a random index from the range that is set.
`0` is the first index position of the array, which in this instance is
Ford.
`makes.length-1` sets the upper limit of the range to be whatever the length
of the `makes` array is, -1 to account for the zero-based indexing.
*/
        model = models[randomIntFromInterval(0,makes.length-1)]
/*
```

```
This works much the same as the above line of code.
`model` is declared as a variable.
`models` is the array declared above, which generated 40 elements of floats
ranging from 1980 to 2019.
`randomIntFromInterval` generates a random index from the range that is set.
`0` is the index of the first postion of the array, which in this case would
be 1980.
`makes.length-1` sets the upper limit of the range to be whatever the length
of the `makes` array is, -1 to account for the zero-based indexing.

*/
```

But I think there's a mistake.....

```
        model = models[randomIntFromInterval(0,makes.length-1)]
/*
`makes.length-1` is making reference to the length of the `makes` array,
which only contains 3 elements. This means there are only 3 possible
answers, meaning that the only 3 options you'll see for a `model` value are
`1980`, `1981`, or `1982`.
*/
        model = models[randomIntFromInterval(0,models.length-1)]
/*
This is code will now allow for a random index our of the full 40 options
available in the `models` array.
*/
```

Last part of the for...of loop:

```


        mycar = new Model(make, model);
/*
`mycar` is declared as a variable that will store the instance of the
`Model` class we're about to create.

`new` creates a new instance of the `Model` class using `make` and `model`
as arguments.

The `Model` constructor which contains `super(brand)` calls the parent
constructor of `Car`, which generates `make` as the `brand`. This then sets
the `carname` property of the instance and generates the string 'I have a '
+ this.carname, which is then stored in the method `present()`.
`this.model = mod` is where the randomly selected element from the `models`
array is stored in the `mod` variable.
```

```
`make` randomly selects an element from the `makes` array of ["Ford",
"Holden", "Toyota"]

`model` randomly selects an element from the `models` array, which contains
every number between 1980 and 2019.
*/
        console.log(mycar.show())
/*
This calls the `show()` method from the `mycar` instance that we created and
prints it as a string to the console.
`show()` gets the string that is stored in `present()` and concatinates it
with  ', it was made in ' and 'this.model`
`this.model` refers to `mod`, which is where the randomly selected element
from the array containing all the numbers between 1980-2019 is stored.
This is all turned into a string.
`console.log` prints this string to console in the browser.
*/
}
```

An example of a string that could be printed to the console would be:

```
"I have a Ford, it was made in 1984"
```
