

## Evidence Gathering Document for SQA Level 8 Professional Developer Award.

This document is designed for you to present your screenshots and diagrams relevant to the PDA and to also give a short description of what you are showing to clarify understanding for the assessor.

Fill in each point with screenshot or diagram and description of what you are showing.

Each point requires details that cover each element of the Assessment Criteria, along with a brief description of the kind of things you should be showing.

### Week 1

Unit	Ref	Evidence	
I&T	I.T.6	Demonstrate the use of a hash in a program. Take screenshots of: *A hash in a program *A function that uses the hash *The result of the function running	

### Paste Screenshot here

#### Screenshot 1

```
hashes_and_functions_more_practice.rb
1 # Nested hash structure, containing the female and male tennis champions, with a few pieces of data about them. The function returns
  * the name of the
2
3 champions_hash = {
4   female_champ: {name: "Serena Williams", total_wins: 23, age: 37},
5   male_champ: {name: "Roger Federer", total_wins: 20, age: 37}
6 }
```

#### Screenshot 2

```
8 def get_female_champion_name(champions_hash)
9   return champions_hash[:female_champ][:name]
10 end
11 p get_female_champion_name(champions_hash)
12
```

#### Screenshot 3

```
week_01_practice — user@users-MacBook-Pro — ..k
→ week_01_practice ruby hashes_and_functions_more_practice.rb
"Serena Williams"
→ week_01_practice
```

### Description here

Screenshot 1 shows a hash, `champions_hash`. It contains two hashes, one for the `female_champ` and one for the `male_champ`. Each of these nested hashes contains key:value pairs that use hash symbols.

Screenshot 2 shows a function I have defined. It returns the name (value) of the `female_champ` hash. The expected result is the string "Serena Williams". Line 11 in Screenshot 2 calls the function using a print command (so that the result can be seen). Given that this is a short and simple program, only one parameter is required (line 8), and the argument that has been called on line 11 is the same as the parameter. Screenshot 3 shows the result of running the file on the terminal: the expected string is shown.

## Week 2

Unit	Ref	Evidence	
I&T	I.T.5	Demonstrate the use of an array in a program. Take screenshots of: *An array in a program *A function that uses the array *The result of the function running	

**Paste Screenshot here**

Screenshot 1

```

pda_array.rb
1  #finding and sorting items from an array
2
3  players = [
4    {name: "Bob",
5      level: "beginner"
6    },
7    {name: "Sue",
8      level: "master"
9    },
10   {name: "Karl",
11     level: "beginner"
12   },
13   {name: "Jenny",
14     level: "beginner"
15   },
16   {name: "Salah",
17     level: "master"
18   }
19 ]
20
21 def find_masters(array)
22   masters = []
23   for player in array
24     masters.push(player) if player[:level] == "master"
25   end
26   return masters
27 end
28
29 p find_masters(players)

```

## Screenshot 2

```

[→ week_02 git:(master) × ruby pda_array.rb
[{:name=>"Sue", :level=>"master"}, {:name=>"Salah", :level=>"master"}]
→ week_02 git:(master) ×

```

## Description here

### Screenshot 1

The file `pda_array.rb` contains an array called `players`. The items in `players` are hashes. Each hash represents a player. Each hash has two key:value pairs, one for the player's name and one for the player's level.

The function `find_masters` searches through an array of players and finds any player that has the level "master". The function requires a user to give an array name as a parameter. To do this, the function creates an empty array, called `masters`. A for loop is then run, searching through the array for any player that has the `:level` key set to "master". If the player meets the condition (has `:level` key set to "master"), the player hash is inserted into the `masters` array. Once the loop has completed, the function returns the `masters` array. An example has been created to print the results of calling the function on the `players` array.

### Screenshot 2

The terminal shows the result of calling the function on the `players` array. There are two items that fit the criteria described in the `find_masters` function.

## Week 3

Unit	Ref	Evidence	
I&T	I.T.3	Demonstrate searching data in a program. Take screenshots of: *Function that searches data *The result of the function running	

### Paste Screenshot here

#### Screenshot 1

```
member.rb
33
34 #define a function to view all members (the READ of CRUD).
35
36 def self.view_all()
37   sql = "SELECT * FROM members
38   ORDER BY last_name"
39   member_data = SqlRunner.run(sql)
40   result = member_data.map { |member| Member.new(member) }
41   return result
42 end
```

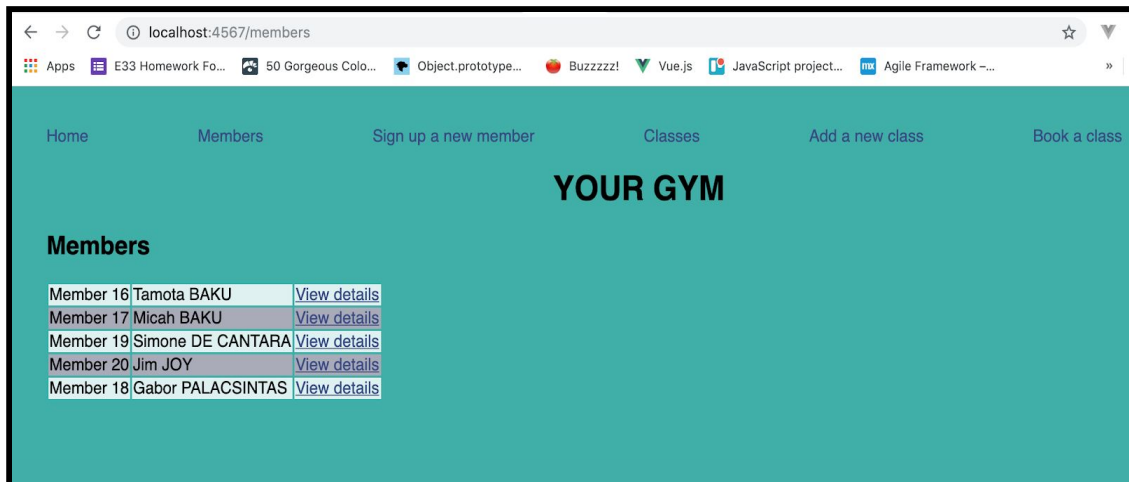
#### Screenshot 2

```
members_controller.rb
25 # show all members
26 get '/members' do
27   @members = Member.view_all
28   erb(:"members")
29 end
```

#### Screenshot 3

```
index.erb
1 <h2>Members</h2>
2
3 <div class="table-container">
4   <table>
5     <% @members.each do |member| %>
6       <tr>
7         <td>Member <%= member.id %></td>
8         <td><%= member.pretty_name %></td>
9         <td><a href="/members/<%= member.id %>">View details</a></td>
10      </tr>
11    <% end %>
12  </table>
13 </div>
```

#### Screenshot 4



### Description here

Screenshot 1 shows the model in which the function `.view_all` is defined. The function will be called the entire Member class. The function goes into the members table of an SQL database and selects all the information on all the items in that table. It then sorts these by the `last_name` property, using the `ORDER BY` statement.

Screenshot 2 shows the function being called on the Member class, saved under a placeholder `@members`.

Screenshot 3 shows the HTML that shows how to display the information from `@members`.

Screenshot 4 shows the finished result in the browser. The searched data is displayed in a table.

Unit	Ref	Evidence	
I&T	I.T.4	Demonstrate sorting data in a program. Take screenshots of: *Function that sorts data *The result of the function running	

### Paste Screenshot here

#### Screenshot 1

```

fitness_class.rb

def self.view_upcoming_classes()
  sql = "SELECT * FROM fitness_classes
  WHERE datetime >= CURRENT_TIMESTAMP
  ORDER BY datetime ASC"
  fitness_classes_data = SqlRunner.run(sql)
  result = fitness_classes_data.map { |fitness_class| FitnessClass.new(fitness_class) }
  return result
end

```

#### Screenshot 2

```

fitness_classes_controller.rb      fitness_class.rb
6 # index
7 get '/fitness_classes' do
8   @upcoming_fitness_classes = FitnessClass.view_upcoming_classes
9   @archived_fitness_classes = FitnessClass.view_archived_classes
10  erb(:"fitness_classes/index")
11 end

```

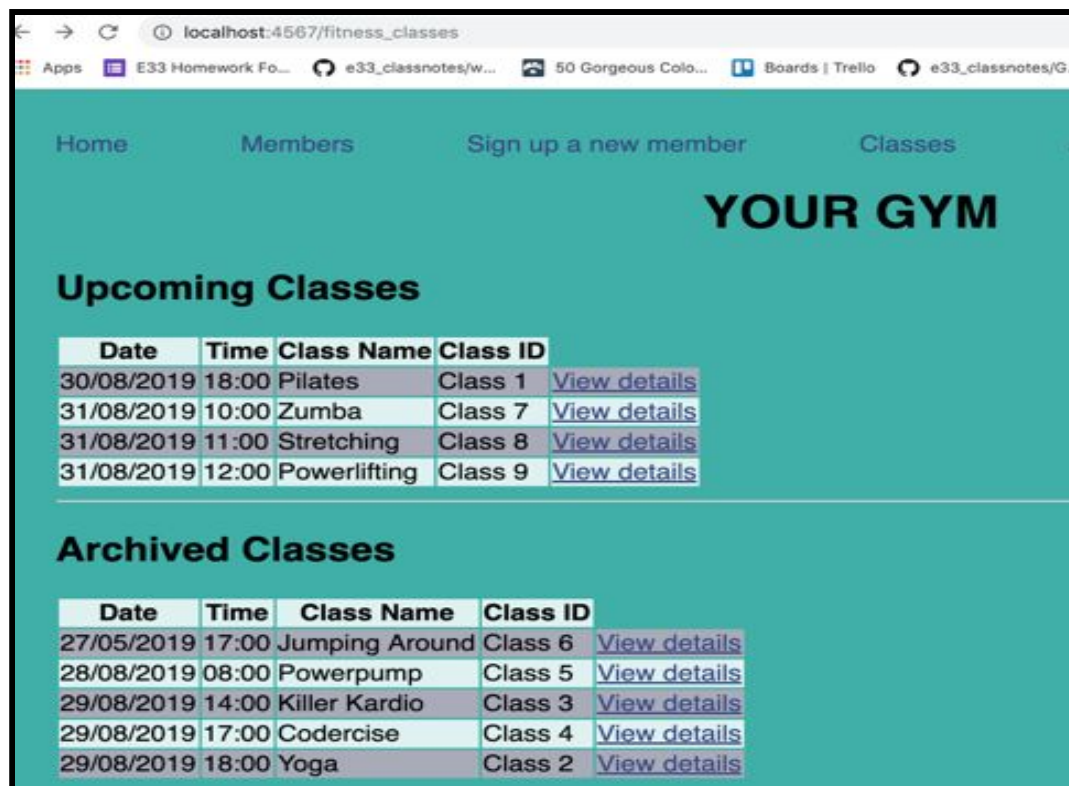
Screenshot 3

```

index.erb
1 <h2>Upcoming Classes</h2>
2
3 <div class="table-container">
4   <table>
5     <tr>
6       <th>Date</th>
7       <th>Time</th>
8       <th>Class Name</th>
9       <th>Class ID</th>
10    </tr>
11    <%= @upcoming_fitness_classes.each do |fitness_class| %>
12      <tr>
13        <td><%= fitness_class.format_date %></td>
14        <td><%= fitness_class.format_time %></td>
15        <td><%= fitness_class.name %></td>
16        <td>Class <%= fitness_class.id %></td>
17        <td><a href="/fitness_classes/<%= fitness_class.id %>">View details</a></td>
18      </tr>
19    <%= end %>
20  </table>
21 </div>

```

Screenshot 4



**Description here**

Screenshot 1 shows the definition of a function in the model. It is called `view_upcoming_classes` and should be called on the class (FitnessClass). The function accesses an SQL table (`fitness_classes`), finds a selection of items that meet the required criterion (`SELECT * FROM fitness_classes WHERE datetime >= CURRENT_TIMESTAMP`), then sorts the results by datetime in ascending order (`ORDER BY date time ASC`).

Screenshot 2 shows the code to call the `view_upcoming_classes` function.

Screenshot 3 shows how the function is placed into an HTML file, so that it will display on screen.

Screenshot 4 shows the result of calling the function. The table `UPCOMING CLASSES` is arranged by date and time in ascending order (that is, oldest to newest).

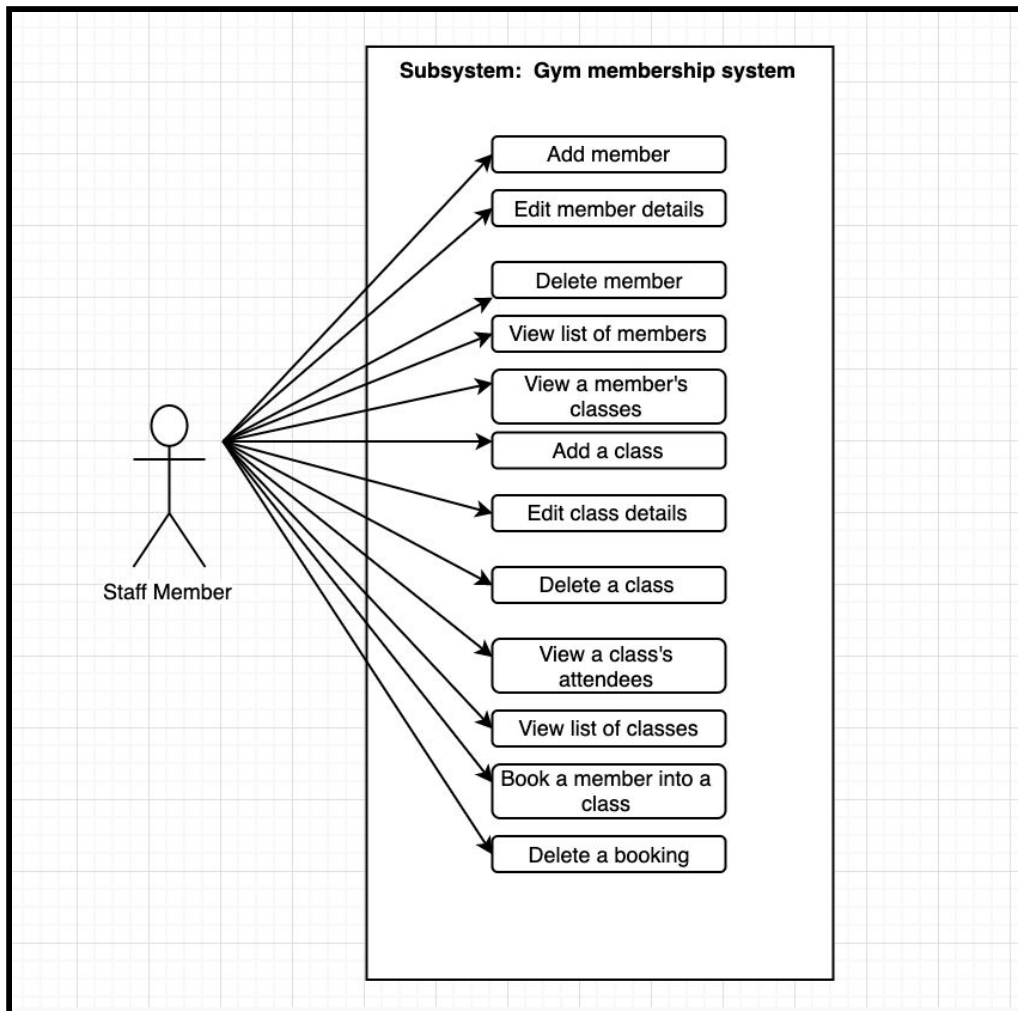
**Week 4**

Unit	Ref	Evidence	
A&D	A.D.1	A Use Case Diagram	

**Paste Screenshot here**

Screenshot of Use Case Diagram





### Description here

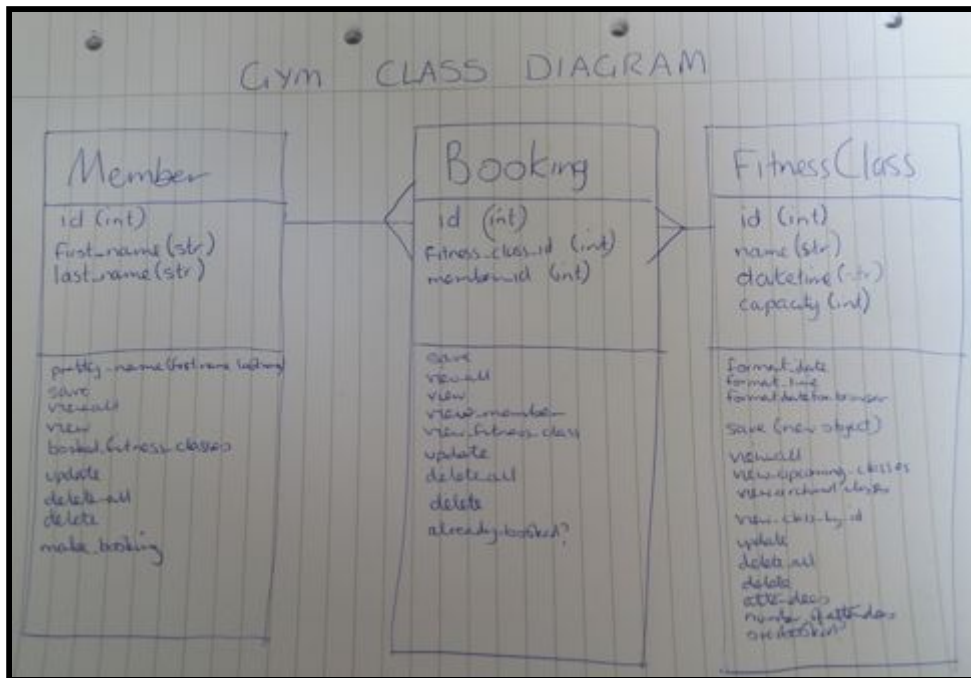
On the left side of the Use Case Diagram is the staff member. On the right side of the diagram is the gym, which is the system that the user is interacting with. In the centre of the diagram is the subsystem that allows the user to interact with the gym. The subsystem is the “membership, classes and booking management system”. The tasks that are carried out by the subsystem are listed, and the lines joining these tasks with the user and the gym indicate that the user is carrying out these tasks and they have an impact on the gym. For example, the user can carry out the task “add member”, which adds a member to the gym.

Unit	Ref	Evidence	
A&D	A.D.2	A Class Diagram	

### Paste Screenshot here

Screenshot of class diagram:





### Description here

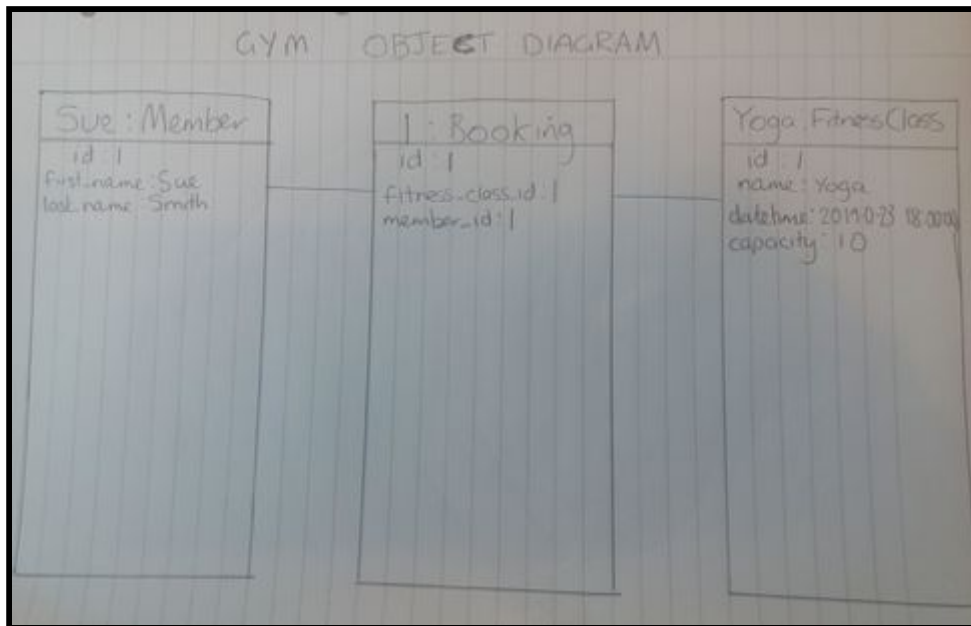
The class diagram shows three classes: Member; Booking; and FitnessClass. I originally identified two classes: Member and FitnessClass. The relationship between these two classes was many to many. That is, an instance of the Member class could be in many instances of the FitnessClass class, and an instance of the FitnessClass class could have many members. So, I created a third class - Booking - to join the first two classes together.

The class diagram shows the properties for each class (for example, the Member class has a property called first\_name). For each property, the diagram shows the data type (for example, first\_name is a string, whilst id is an integer).

In terms of functions, all classes would need CRUD functionality (that is, the ability to Create, Read, Update and Delete instances of the respective class). There were additional functions that I anticipated would be required. For example, the FitnessClass class has a method called "overbooked?", which would be called when a new booking was being made. If the class capacity had been reached before the booking was made, then the app would prevent the booking from being made. The functions in the diagram were based on my initial planning and are subject to change as the project develops.

Unit	Ref	Evidence	
A&D	A.D.3	An Object Diagram	

Paste Screenshot here

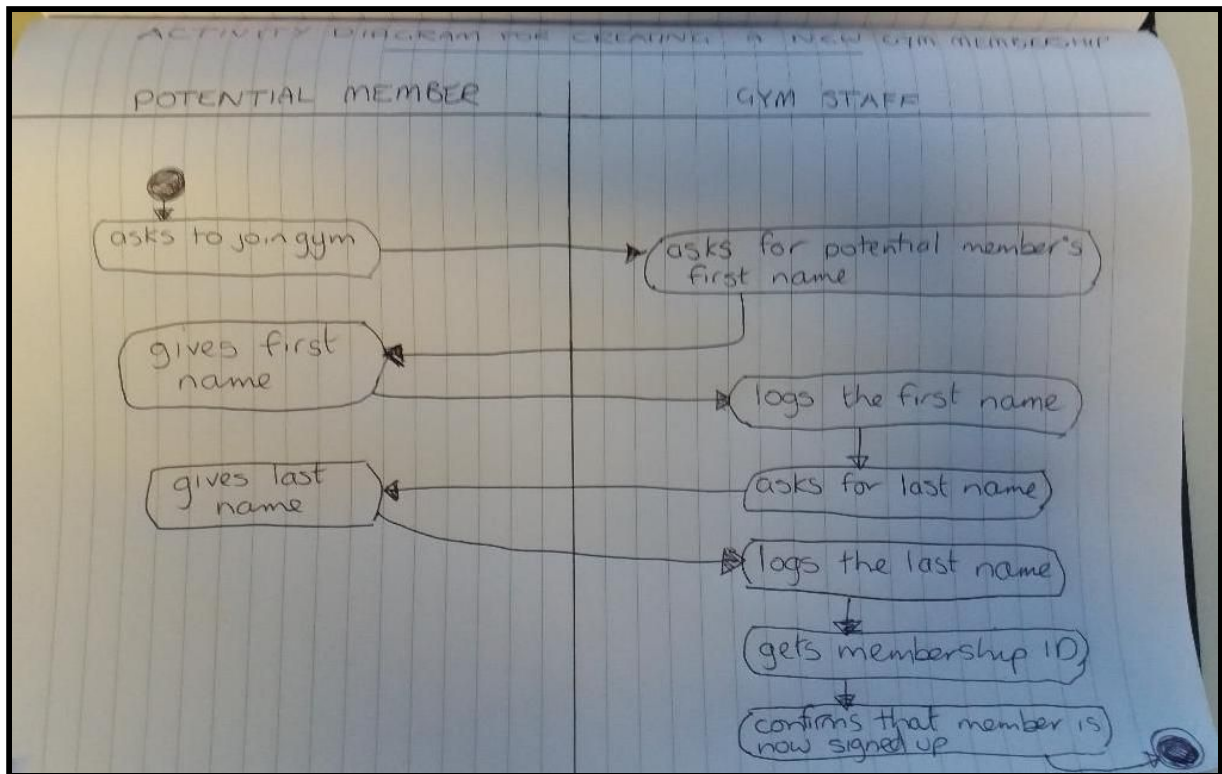


### Description here

There are three instances of classes: Sue Smith (an instance of the Member class); booking1 (an instance of the Booking class); and Yoga (an instance of the FitnessClass class). Since all of these are the first instances of their respective classes, they each have a value of 1 in the id property. Because booking1 also takes the ids from Sue Smith and yoga, it has member\_id 1 and fitness\_class\_id 1. Yoga also takes a datetime value and capacity value.

Unit	Ref	Evidence	
A&D	A.D.4	An Activity Diagram	

Paste Screenshot here



### Description here

The Activity Diagram shows the process for creating a new gym membership. First, the potential member asks to join the gym. Then the gym staff gets relevant details from the potential member (in this case, the first and last name are the only details that are required). Next, the gym staff adds these details to the gym's members list. This list will show a member ID. That is, if this potential member is the first person to ask for membership, their ID would be 1, the second person would be 2, and so on). If there are no problems with this, the process is successful, and the potential member is now a member.

Unit	Ref	Evidence	
A&D	A.D.6	Produce an Implementations Constraints plan detailing the following factors: *Hardware and software platforms *Performance requirements *Persistent storage and transactions *Usability *Budgets *Time	

Paste Screenshot here

Constraint Category	Implementation Constraint	Solution
Hardware and Software Platforms	<p>Users might be using browsers/software/devices that are different from the one I created the software on.</p> <p>This could result in the app displaying differently for different users, and could result in a display that is ugly or difficult to use.</p> <p>This could lead to: the client may refuse to buy the product, or may switch to another software provider, resulting in lost business to our company.</p>	<p>Test the app on a range of browsers and devices. Use Google Developer Tools to see how the app looks on mobile device.</p>
Performance Requirements	<p>The user's system might not be fast/powerful enough to run all the features of the app.</p> <p>This could lead to problems such as that pictures might not load or might load slowly. This could slow down the user, could prevent them from using all the apps features, or might lead to additional costs for them (such as purchasing faster hardware).</p> <p>The client would not want to use the product and might switch software provider.</p>	<p>Have alt text for pictures.</p>
Persistent Storage and Transactions	<p>There may be insufficient space to hold all the data.</p> <p>This could mean that the user is not able to put all members or classes into the database.</p>	<p>For data items (gym users) that have not used the database for a long time (let's say a year), the items could be archived.</p>

	<p>This could lead to a loss of members for the client, which could lead to the client going out of business or switching to another software provider.</p>	
Usability	<p>The client wants something that is simple to use.</p> <p>This is so that users can learn to use the software with a small amount of training. The client has a high number of users: in the case of my project the client was a gym, and the users would be gym staff. The gym employs many staff, often on a short-term or casual basis.</p> <p>If the software is too complex to use, it increases the client's costs and could lead them to switch software provider.</p>	Focus on the user experience.
Budgets	<p>There is not much money for this.</p> <p>This means that our business cannot devote significant resources in terms of time, money or staff.</p> <p>The more resources we devote to the project, the higher the cost to the client, who might look to another software provider if the cost of our software is too high.</p>	Make the app fairly basic. Rather than getting outside contractors to work on the HTML and CSS, keep this in-house and make the HTML and CSS basic.
Time Limitations	<p>The client wants working software as soon as possible.</p> <p>This limits the amount of time available for developing the app and means that the technologies and languages used should not be too complex or require a lot of research. If we wait until every</p>	Get the app to Minimum Viable Product quickly, so that the user can begin to use it as soon as possible, and make extensions after this. The Agile methodology would be suitable for this project.

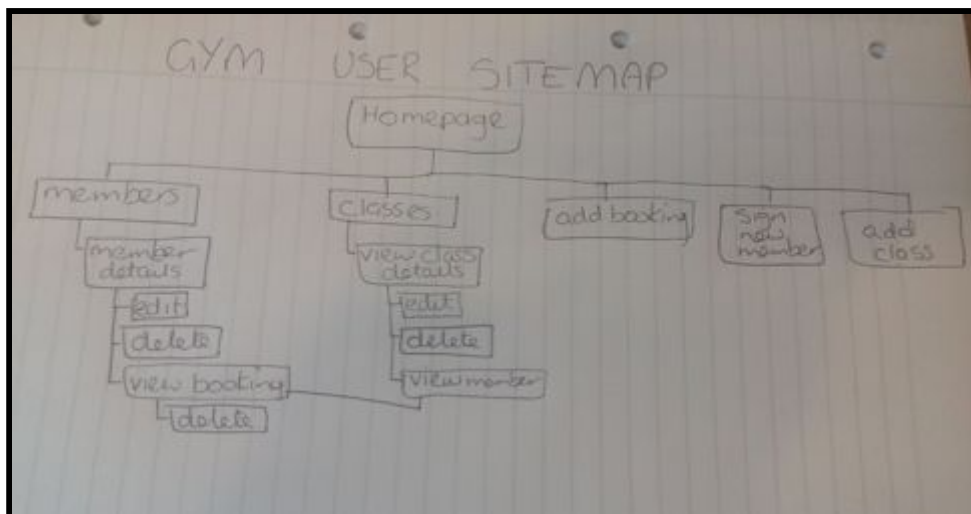
	<p>feature is complete before we deliver the app, the client would not be happy. Moreover, if we waited until every feature is complete and only then discovered that one or more feature does not meet the client's needs, rework would be required.</p> <p>This would mean not only that we had a dissatisfied client (who may not engage our services in the future), it also means we would have extra work to do, increasing our costs.</p>	
--	--	--

### Description here

The Implementation Constraints Plan outlines anticipated constraints for a project to create an app, and shows solutions for these constraints.

Unit	Ref	Evidence	
P	P.5	User Site Map	

### Paste Screenshot here



### Description here

Reading from top to bottom, we can see that there is a **homepage** (the parent element). It has five child elements: **members**; **classes**; **add booking**; **sign new member**; and **add class**. The **members** and **classes** in turn have their own child elements: respectively, **member details** and **view class details**. **Member details** and **view class details** also have

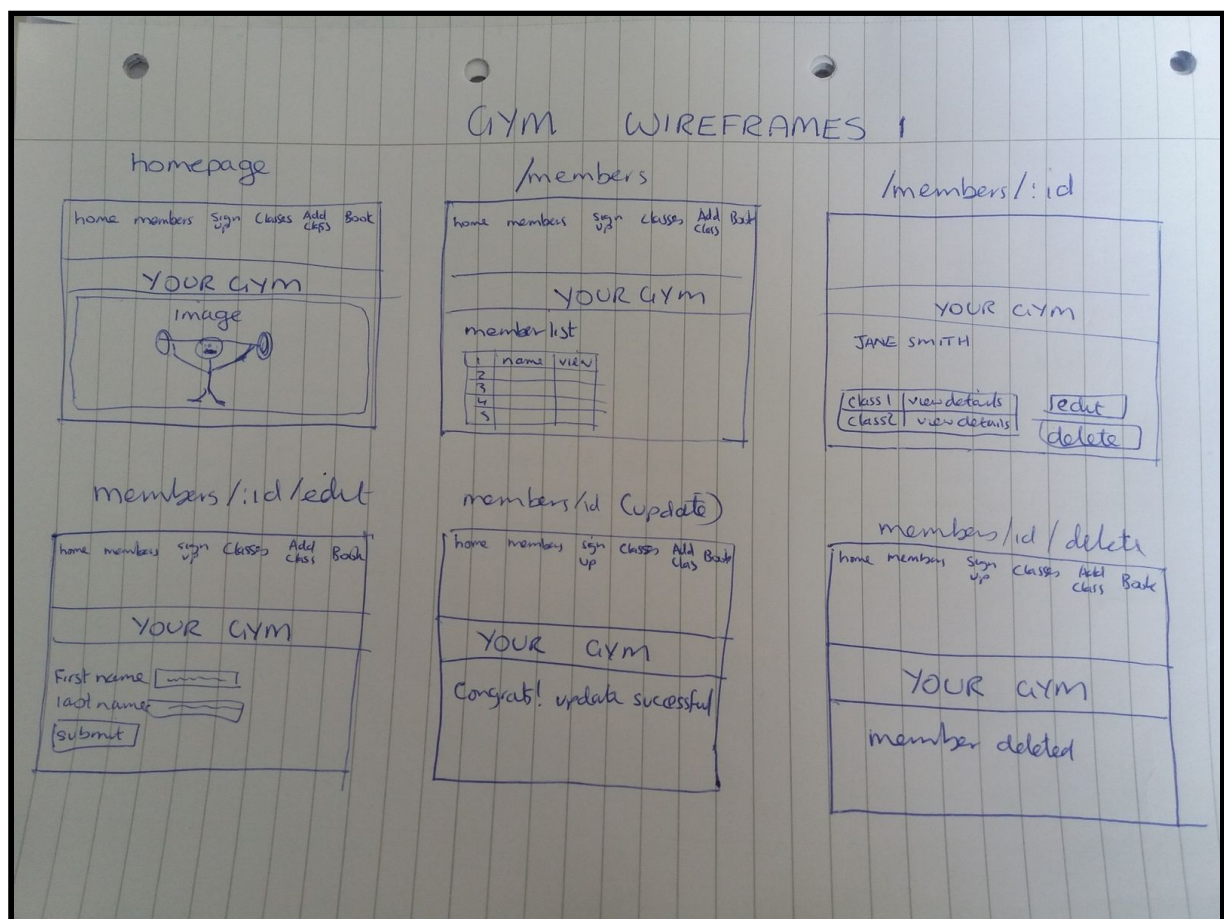
child elements. Member details has three child elements: **edit**; **delete**; and **view booking**. **View booking** has a child element called **delete**. **View class details** also has **edit** and **delete** child elements, plus **view member**. This **view member** element contains a link to the aforementioned **view booking**.

The **add booking**, **sign new member**, and **add class** elements do not have child elements.

Unit	Ref	Evidence	
P	P.6	2 Wireframe Diagrams	

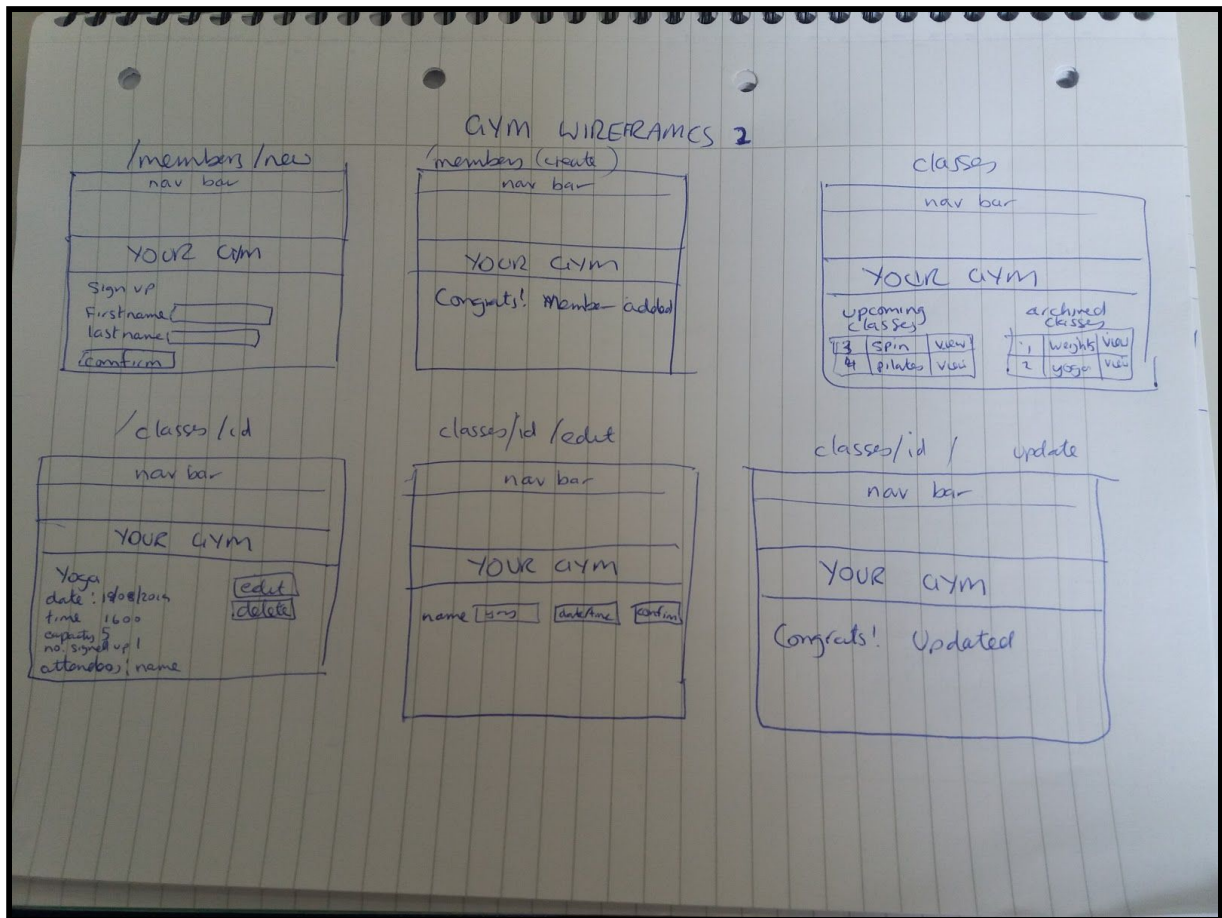
Paste Screenshot here

Screenshot 1



Screenshot 2





### Description here

Screenshot 1 shows six wireframes that take the user through the process of viewing, editing and deleting members.

In Screenshot 2, the first two wireframes are for adding a new member to the members list. The next four wireframes show how to view, edit and delete gym classes.

### Week 5

Unit	Ref	Evidence	
P	P.10	Example of Pseudocode used for a method	

Paste Screenshot here

```

113 # method to stop a member being booked into the same fitness_class multiple times. It will be
114 # called on the Booking class. First, we want to check whether that member is in the list of
115 # attendees for a particular class, so it will need to take the member_id and the fitness_class_id.
116 # An sql query will be run to select any items in the database that have the member_id and
117 # fitness_class_id. If the result of that query is not nil, then return.
118 def self.already_booked?(member_id, fitness_class_id)
119   sql = "
120     SELECT * FROM bookings
121     WHERE member_id = $1 AND fitness_class_id = $2
122   "
123   values = [member_id, fitness_class_id]
124   no_booking = SqlRunner.run(sql, values).first.nil?
125   return !no_booking
126 end

```

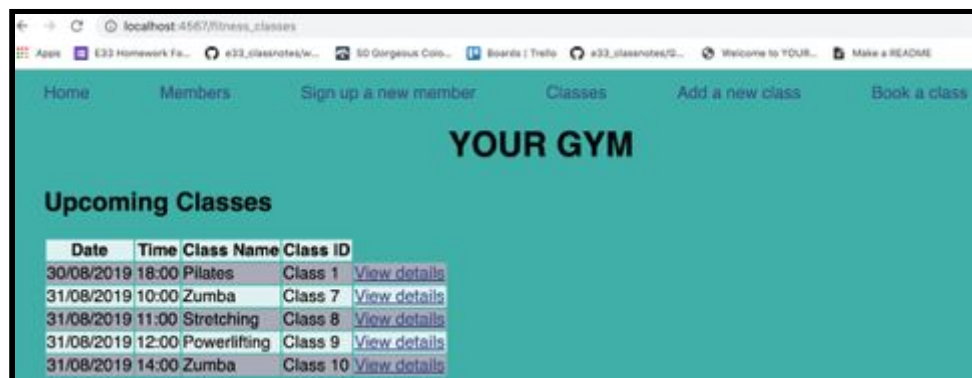
## Description here

The pseudocode is for a function that will prevent a user from booking a member into the same fitness class multiple times.

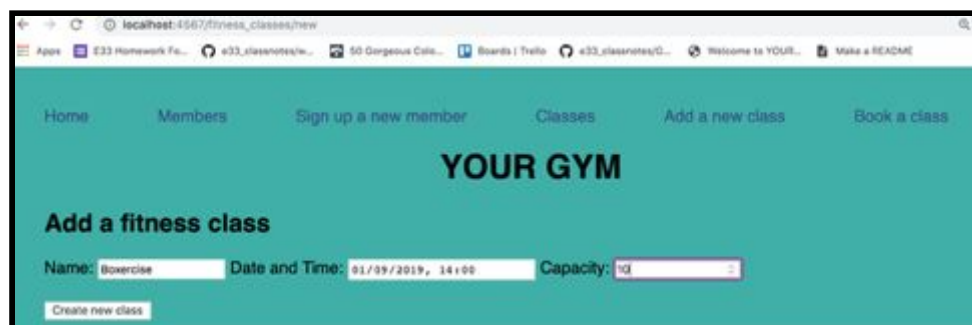
Unit	Ref	Evidence	
P	P.13	<p>Show user input being processed according to design requirements. Take a screenshot of:</p> <ul style="list-style-type: none"> <li>* The user inputting something into your program</li> <li>* The user input being saved or used in some way</li> </ul>	

## Paste Screenshot here

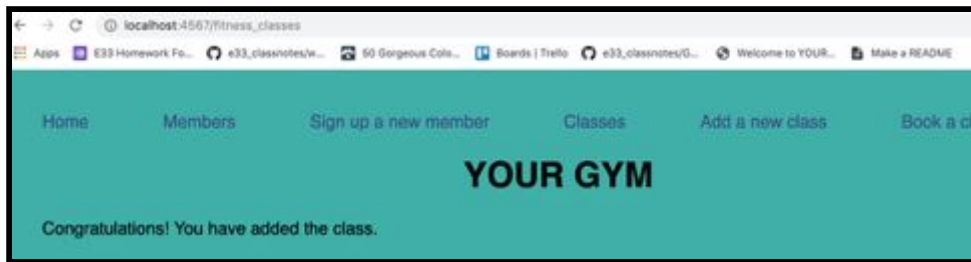
Screenshot 1: upcoming classes before user input



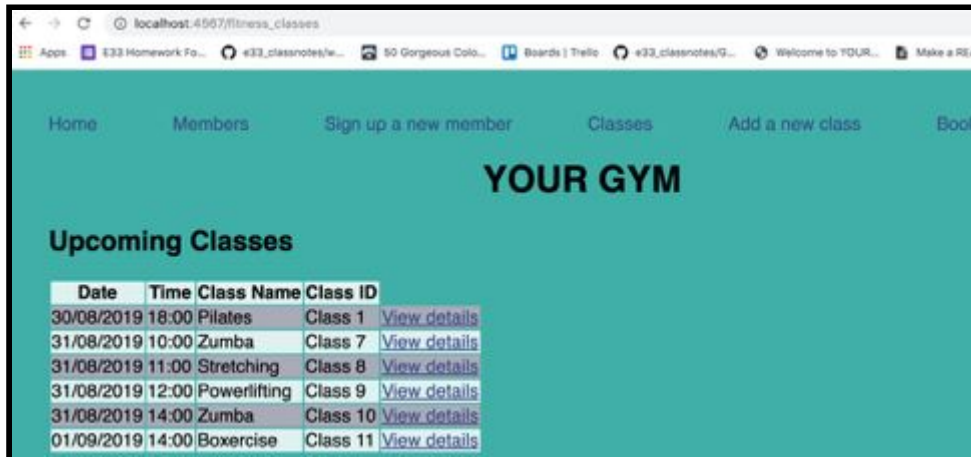
Screenshot 2: user input



Screenshot 3: confirmation screen shows user input is successful



Screenshot 4: upcoming classes after user input



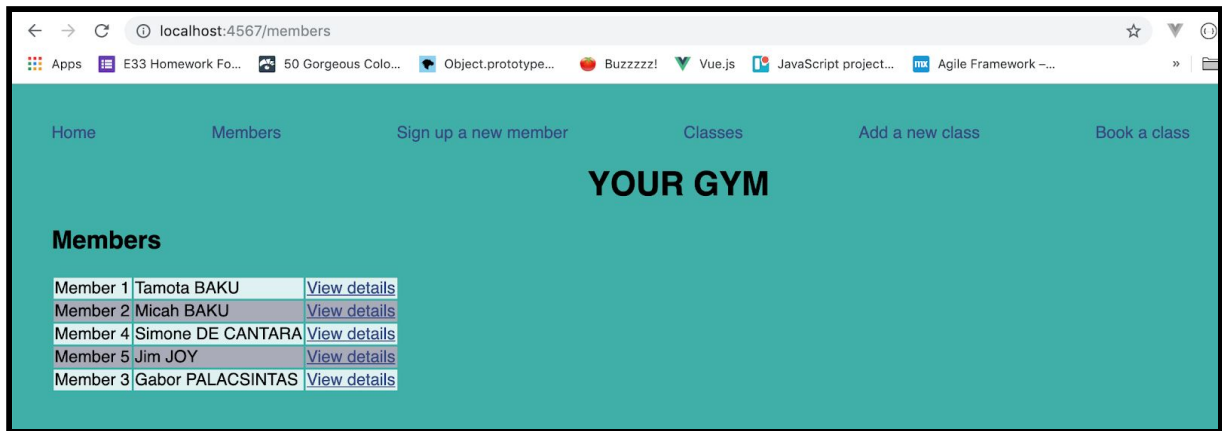
### Description here

Screenshots 1-4 show the user's journey through the process of adding an upcoming fitness class. Screenshot 1 shows the list of upcoming classes before the user's input. There are five upcoming fitness classes. Screenshots 2 and 3 show the process of adding the class. Screenshot 4 shows that the user's action has been successful, and that a new fitness class (Boxercise) has been added to the list of upcoming fitness classes.

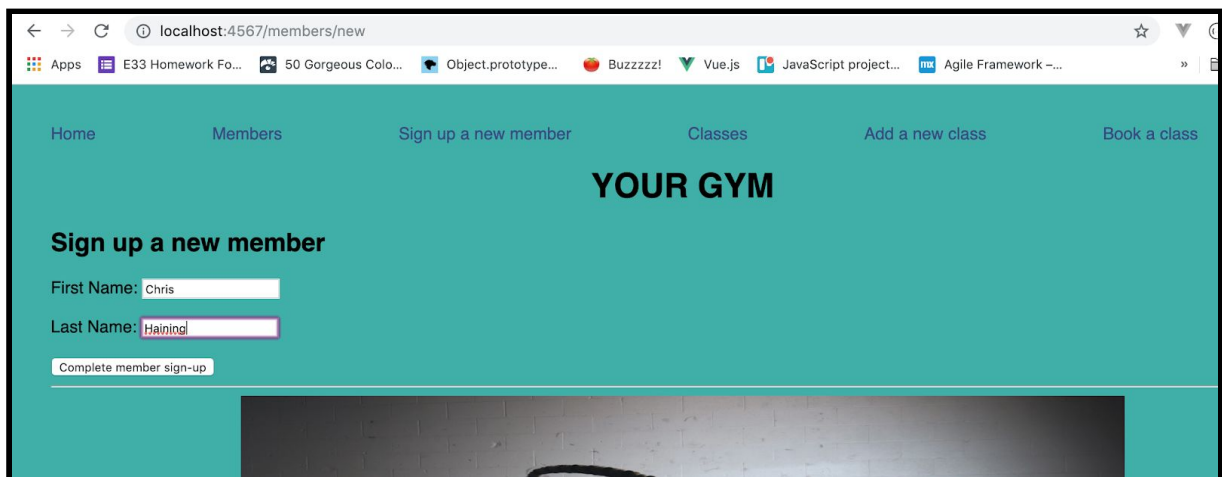
Unit	Ref	Evidence	
P	P.14	Show an interaction with data persistence. Take a screenshot of: <ul style="list-style-type: none"> <li>* Data being inputted into your program</li> <li>* Confirmation of the data being saved</li> </ul>	

### Paste Screenshot here

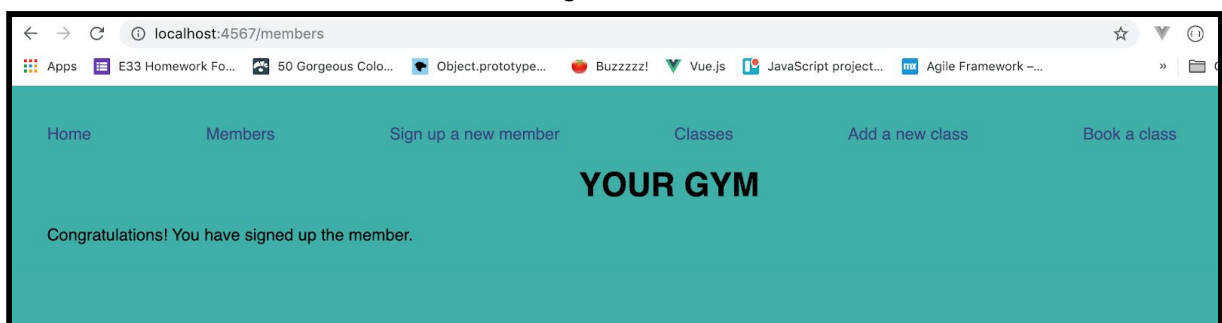
Screenshot 1: the list before the data is inputted into the program



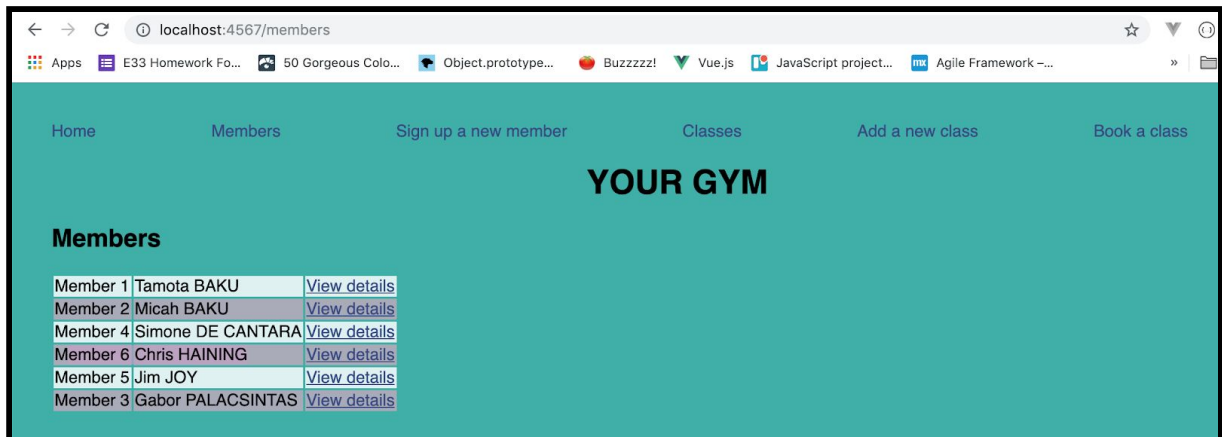
Screenshot 2: data being inputted into the program



Screenshot 3: confirmation of the data being saved



Screenshot 4: the updated list



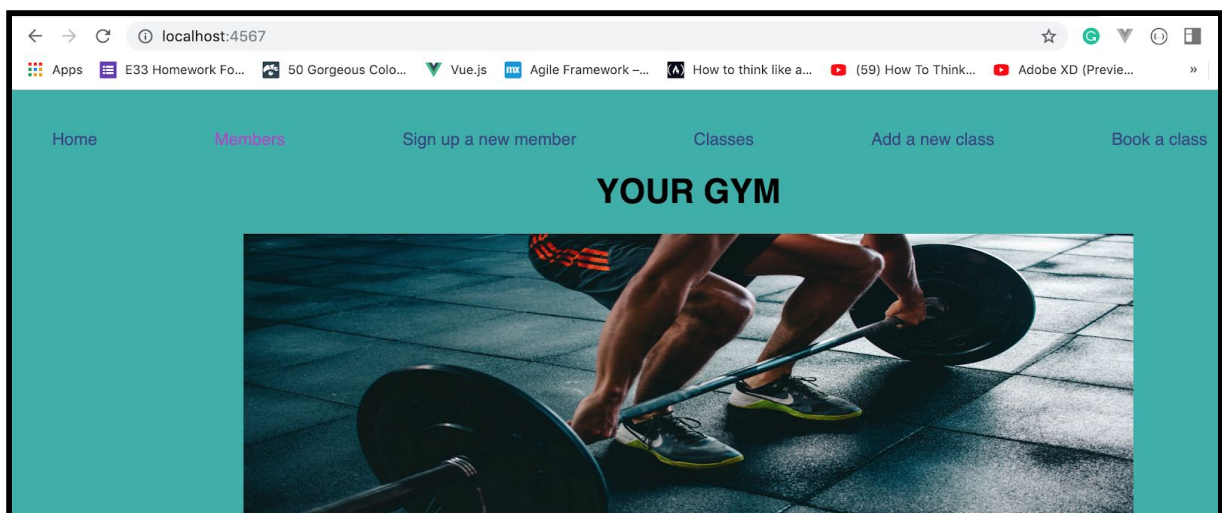
### Description here

The screenshots 1-4 show the Members list being updated. The original list contains five members (Screenshot 1). The user then inputs a new member's name (Screenshot 2). Screenshot 3 gives a message telling the user that the new member has been added, and Screenshot 4 provides further confirmation that the new member has been added by showing an updated list with six members.

Unit	Ref	Evidence	
P	P.15	Show the correct output of results and feedback to user. Take a screenshot of: <ul style="list-style-type: none"> <li>* The user requesting information or an action to be performed</li> <li>* The user request being processed correctly and demonstrated in the program</li> </ul>	

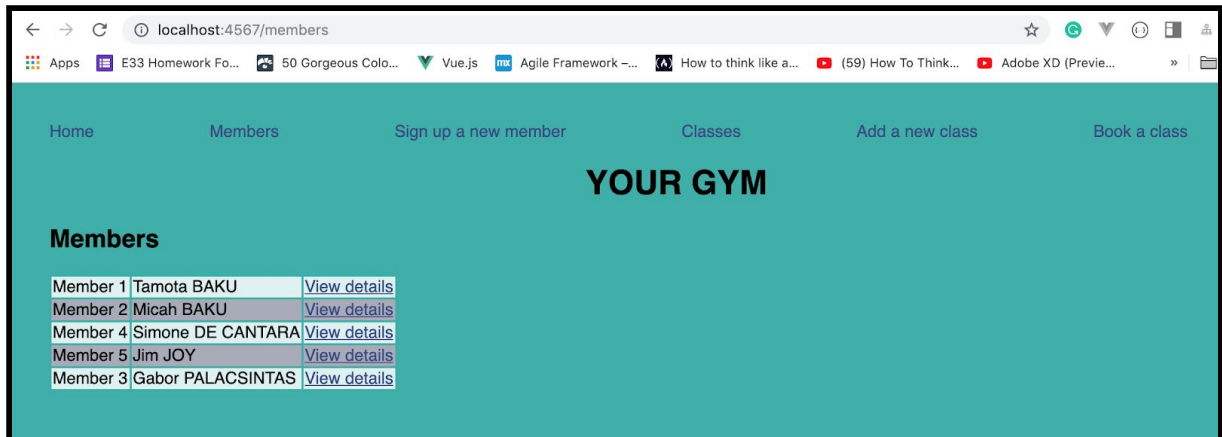
### Paste Screenshot here

#### Screenshot 1





## Screenshot 2



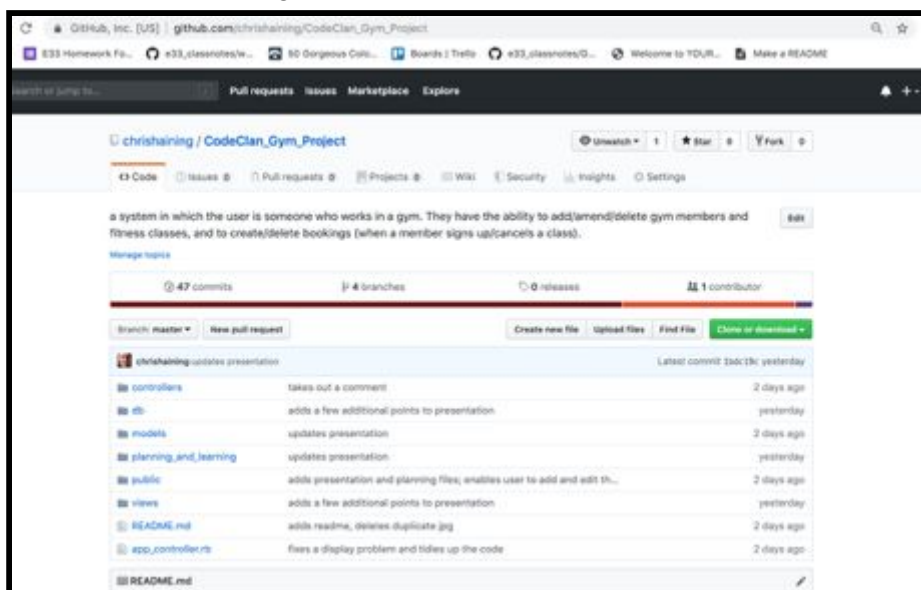
## Description here

Screenshot 1 shows that the user is hovering over the Members item in the navigation bar. This is so that the user can view members. Screenshot 2 shows the result of the user's request: the list of members is now visible.

Unit	Ref	Evidence	
P	P.11	Take a screenshot of one of your projects where you have worked alone and attach the Github link.	

## Paste Screenshot here

Screenshot of working alone on Github



## Description here

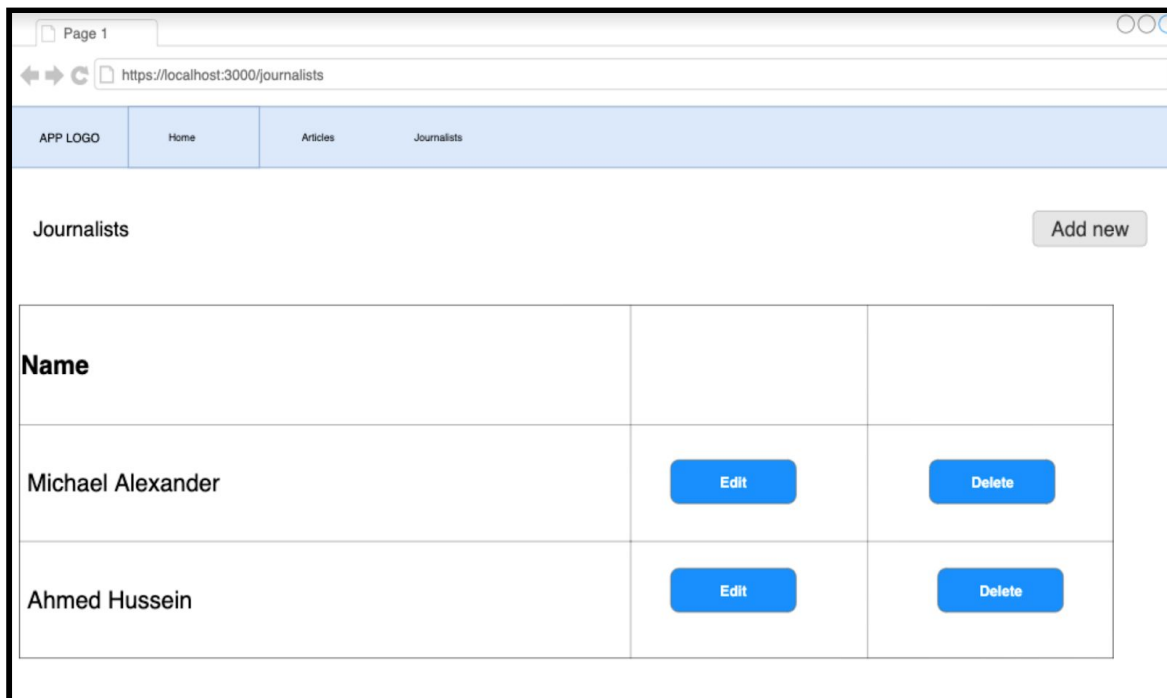
The screenshot shows a link to my solo project, CodeClan\_Gym\_Project.

Link: [https://github.com/chrishaining/CodeClan\\_Gym\\_Project](https://github.com/chrishaining/CodeClan_Gym_Project)

Unit	Ref	Evidence	
P	P.12	Take screenshots or photos of your planning and the different stages of development to show changes.	

## Paste Screenshot here

### Screenshot 1



### Screenshot 2



Page 1

https://localhost:3000/journalists/1

APP LOGO

Home

Articles

Journalists

### Edit a journalist

Last Name

Alexander

First Name

Michael

Phone Number

0123 456 789

Submit

Screenshot 3

Page 1

https://localhost:3000/journalists

APP LOGO

Home

Articles

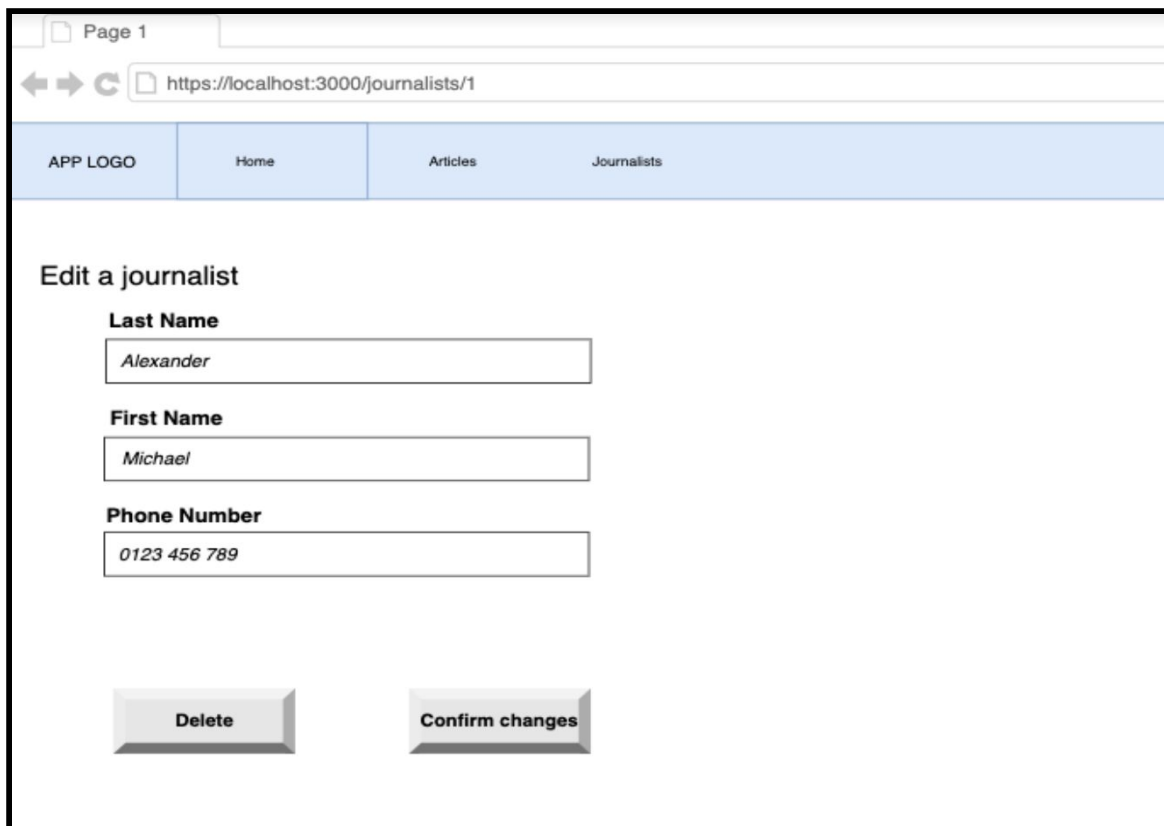
Journalists

Journalists

Add new

Name
<a href="#">Michael Alexander</a>
<a href="#">Ahmed Hussein</a>

## Screenshot 4



The screenshot shows a web browser window with the address bar displaying 'https://localhost:3000/journalists/1'. The browser's page title is 'Page 1'. The application has a light blue navigation bar with four items: 'APP LOGO', 'Home', 'Articles', and 'Journalists'. The main content area is titled 'Edit a journalist'. It contains three text input fields: 'Last Name' with the value 'Alexander', 'First Name' with the value 'Michael', and 'Phone Number' with the value '0123 456 789'. At the bottom of the form are two buttons: 'Delete' and 'Confirm changes'.

### Description here

The four screenshots show a selection of wireframe diagrams I used to plan a news app. Screenshots 1 and 2 are based on the original plan, whilst screenshots 3 and 4 have been revised. In Screenshot 1, the intention was that users would see a table of all the journalists in the database. If the user wanted to edit or delete a journalist, they would click on the relevant button in the table. If the user clicked on the edit button, they would be led to the edit screen (Screenshot 2).

However, during the construction of the app the plan changed. Instead of having separate edit and delete buttons in the table, the user would be able to click on the journalist's name from the table (Screenshot 3). This would take the user to the page of the journalist, and here they would be able to edit or delete the journalist: to edit, the user would type the relevant changes on the form then use the Confirm changes button; to delete, the user would press the delete button. The changes to the wireframes were due to feedback from other students who acted as users. They advised that having the buttons in the table did not look good, and that it seemed risky to delete a journalist without seeing their full details.

## Week 7

Unit	Ref	Evidence	
------	-----	----------	--

P	P.16	Show an API being used within your program. Take a screenshot of: <ul style="list-style-type: none"> <li>* The code that uses or implements the API</li> <li>* The API being used by the program whilst running</li> </ul>
---	------	--

**Paste Screenshot here**

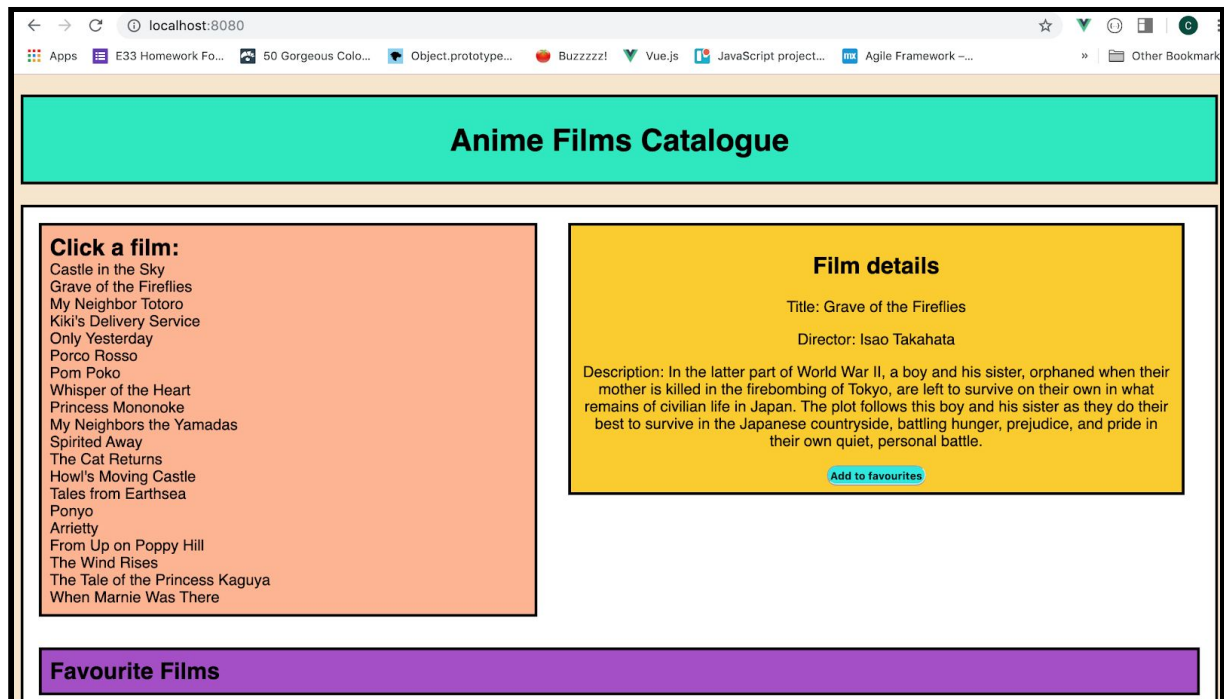
Screenshot 1: The code that uses or implements the API

```

App.vue
12
13 </template>
14
15 <script>
16 import FilmsList from './components/FilmsList'
17 import FilmDetails from './components/FilmDetails'
18 import FavouriteFilms from './components/FavouriteFilms'
19
20 import {eventBus} from './main'
21
22 export default {
23   name: 'app',
24   data() {
25     return {
26       films: [],
27       selectedFilm: null,
28       favouriteFilms: []
29     }
30   },
31
32   mounted() {
33     fetch('https://ghibliapi.herokuapp.com/films')
34       .then(result => result.json())
35       .then(films => this.films = films)
36

```

Screenshot 2: The API being used by the program whilst running



**Description here**

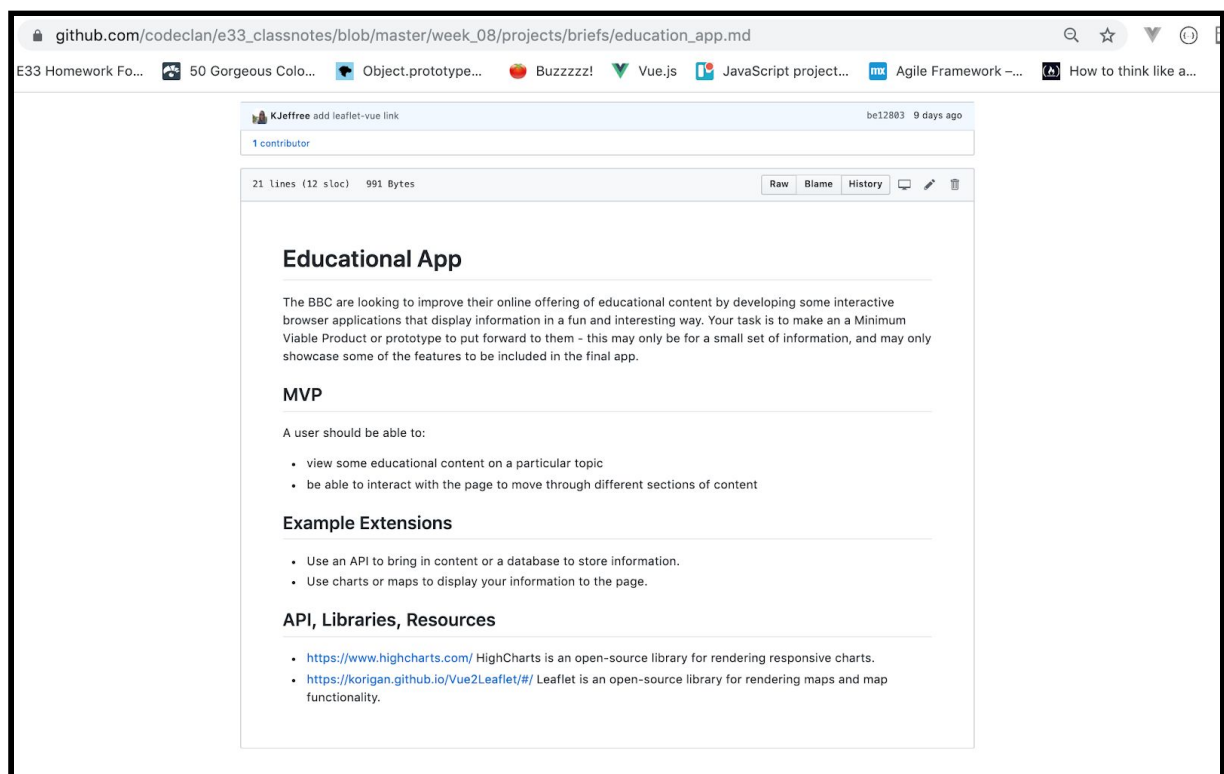
Screenshot 1 shows that the App.vue file has a method to fetch the details from the API ['https://ghibliapi.herokuapp.com/films'](https://ghibliapi.herokuapp.com/films). Screenshot 2 shows the program in action. The full

list of films in the API is shown on the left-hand side under “Click a film:”. The right-hand side shows the details of one of the films. At the bottom is a collection, Favourite Films, which starts empty. If the user clicks on the “Add to favourites” button in the Film Details box, the film will be added to the Favourite Films collection.

## Week 8

Unit	Ref	Evidence	
P	P.2	Take a screenshot of the project brief from your group project.	

## Paste Screenshot here



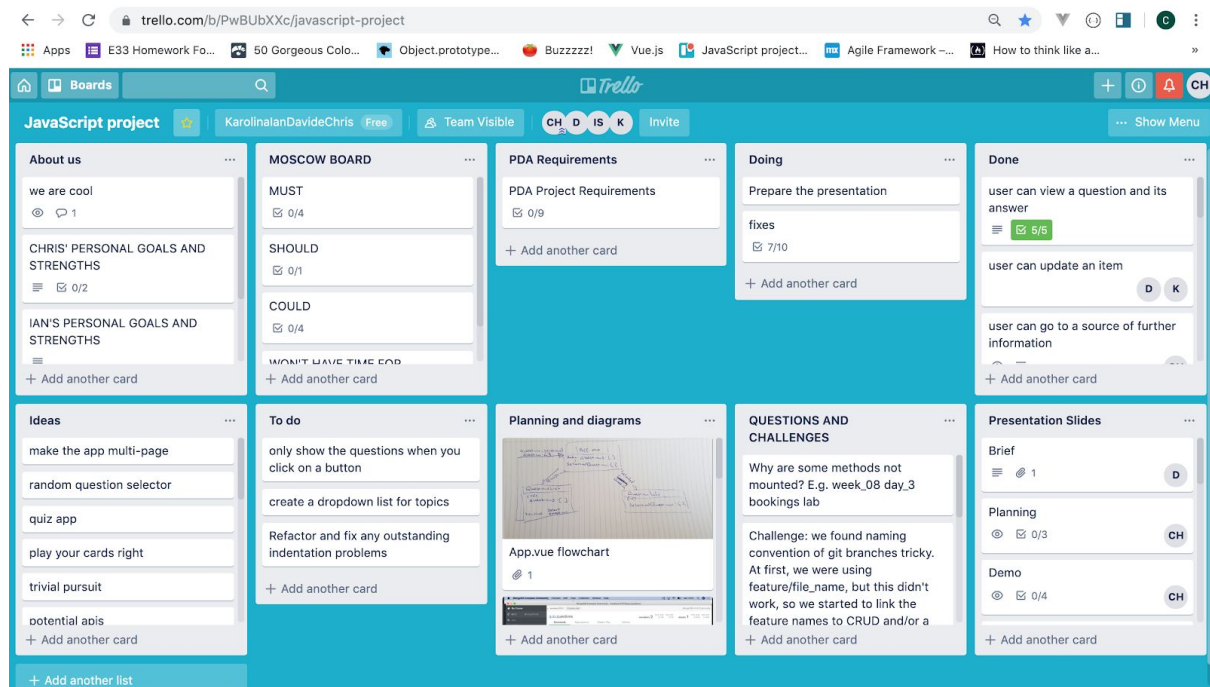
## Description here

This brief includes an overview of who the client is and what they want. It then lists the MVP (Minimum Viable Product) which gives more specific features of what the product (an app) has to do in order to meet the brief. The Example Extensions are optional features. Translated into a MOSCOW board format, these extensions would be considered SHOULD, COULD or WON'T HAVE TIME FOR features, whereas the MVP tells us the MUST features. The API, Libraries, Resources section gives some resources that could help with the extensions.

Unit	Ref	Evidence	
P	P.3	Provide a screenshot of the planning you completed during your group project, e.g. Trello MOSCOW board.	

## Paste Screenshot here

### Screenshot of Trello board



## Description here

The screenshot shows the Trello board for a group project. The project's product is an app. There is a MOSCOW board showing the features that the app **MUST**, **SHOULD** and **COULD** have, as well as the features that might be nice, but that we **WON'T HAVE TIME FOR**. The project team members converted these features into tasks, which are shown on the **TO DO**, **DOING** and **DONE** lists. Each task moves from **TO DO**, through **DOING**, until it is **DONE**.

Unit	Ref	Evidence	
P	P.4	Write an acceptance criteria and test plan.	

## Paste Screenshot here

Criterion: the user can ...	Expected result	Pass/fail
View a list of all members	The list of members appears when the user clicks the Members section of the	pass

	navigation bar.	
Go to the "Sign up a new member" form.	The "Sign up a new member" form appears when the user clicks the "Sign up a new member" section of the navigation bar.	pass
Sign up a new member	A new member is created when the user clicks the "Complete member sign-up" button in the "Sign up a new member" form.	pass
View a list of classes	The list of classes appears when the user clicks the Classes section of the navigation bar.	pass
Go to the "Add a new class" form.	The "Add a new class" appears when the user clicks the "Add a new class" section of the navigation bar.	pass
Add a new class	A new class is created when the user clicks the "Create new class" button in the "Add a new class" form.	pass
Go to the "Book a class" form	The "Book a class" form appears when the user clicks the "Book a class" section of the navigation bar.	pass
Book a class	A booking is created when the user clicks the "Confirm booking" button in the "Book a class" form.	pass

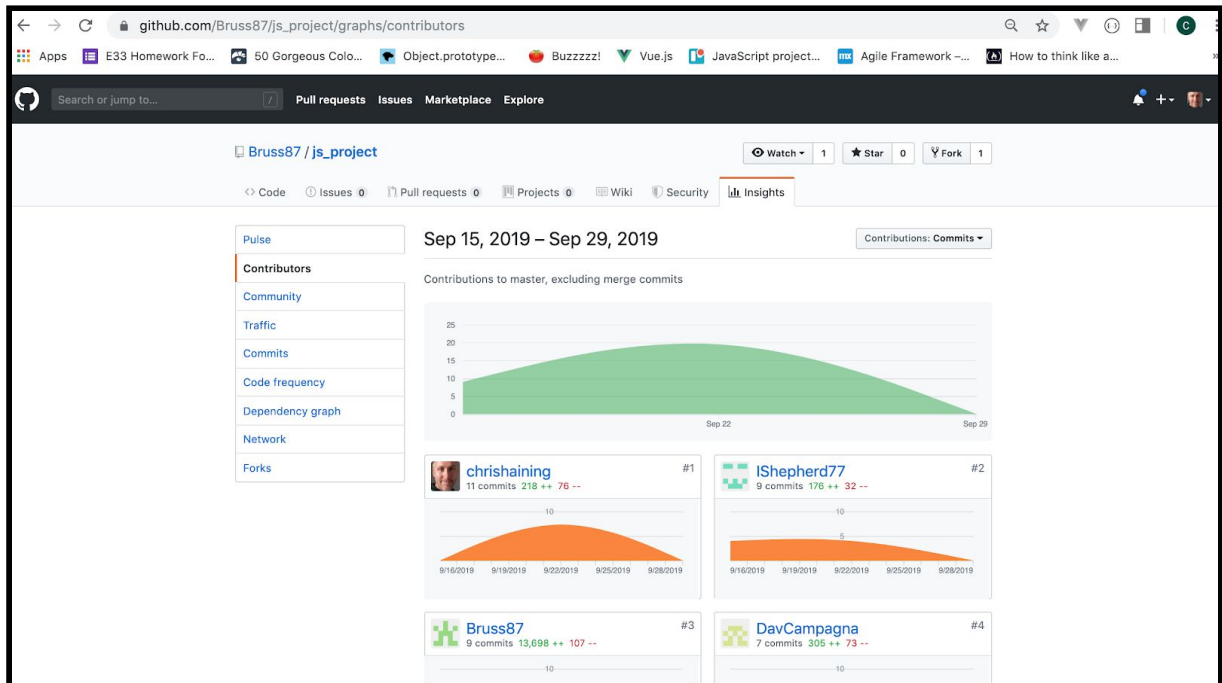
### Description here

The Acceptance Criteria and Test Plan shows all the criteria for my Gym project. These criteria were those we decided were essential to the product (an app). All the criteria passed.

### Week 9

Unit	Ref	Evidence	
P	P.1	Take a screenshot of the contributor's page on Github from your group project to show the team you worked with.	

### Paste Screenshot here



### Description here

[https://github.com/Bruss87/js\\_project/graphs/contributors](https://github.com/Bruss87/js_project/graphs/contributors)

### Week 11

Unit	Ref	Evidence	
P	P.18	Demonstrate testing in your program. Take screenshots of: * Example of test code * The test code failing to pass * Example of the test code once errors have been corrected * The test code passing	

### Paste Screenshot here

Screenshot of test code



```

79
80 // What does the code do in exceptional circumstances? Specifically, if you divide by zero, what is the effect?
81 * Write a test to describe what you'd prefer to happen, and then correct the code to make that test pass (you will
82 * need to modify the Calculator model to meet this requirement).
83 it('is the output as expected for a number divided by zero', function(){
84     running_total = element(by.css('#running_total'))
85     element(by.css('#number9')).click();
86     element(by.css('#operator_divide')).click()
87     element(by.css('#number0')).click();
88     element(by.css('#operator_equals')).click()
89     expect(running_total.getAttribute('value')).to.eventually.equal('ERROR')
90 })

```

## Screenshot of test code failing to pass

```

js_calculator_start_point git:(master) npm run protractor
> js_calculator_start_point@1.0.0 protractor /Users/user/codeclan_work/week_08/Unit_and_Integration_Task_B/js_calculator_start_point
> protractor conf.js

[12:00:02] I/launcher - Running 1 instances of WebDriver
[12:00:02] I/hosted - Using the selenium server at http://localhost:4444/wd/hub

- calculator functionality should have working number buttons
- calculator functionality the number buttons should update the display of the running total
- calculator functionality the arithmetical operations should update the display with the result of the operation
- calculator functionality multiple operations can be chained together
- calculator functionality is the output as expected for a negative number
- calculator functionality is the output as expected for a decimal
- calculator functionality is the output as expected for a very large number
- calculator functionality is the output as expected for a number divided by zero
1) calculator functionality is the output as expected for a number divided by zero

0 passing (555ms)
7 pending
1 failing

1) calculator functionality
   is the output as expected for a number divided by zero:

      AssertionError: expected 'Infinity' to equal 'ERROR'
      + expected - actual
      -Infinity
      +ERROR

From: Task: calculator functionality is the output as expected for a number divided by zero
at Context.ret (node_modules/selenium-webdriver/testing/index.js:182:10)

[12:00:04] I/launcher - 0 instance(s) of WebDriver still running
[12:00:04] I/launcher - chrome #01 failed 1 test(s)
[12:00:04] I/launcher - overall: 1 failed spec(s)
[12:00:04] E/launcher - Process exited with error code 1
npm ERR! code ELIFECYCLE
npm ERR! errno 1
npm ERR! js_calculator_start_point@1.0.0 protractor: `protractor conf.js`
npm ERR! Exit status 1
npm ERR!
npm ERR! Failed at the js_calculator_start_point@1.0.0 protractor script.
npm ERR! This is probably not a problem with npm. There is likely additional logging output above.

npm ERR! A complete log of this run can be found in:
npm ERR! /Users/user/.npm/_logs/2019-10-02T11:00:04.700Z-debug.log

```

## Screenshot of the test code once errors have been corrected

```

79
80 // What does the code do in exceptional circumstances? Specifically, if you divide by zero, what is the effect?
81 * Write a test to describe what you'd prefer to happen, and then correct the code to make that test pass (you will
82 * need to modify the Calculator model to meet this requirement).
83 it('is the output as expected for a number divided by zero', function(){
84     running_total = element(by.css('#running_total'))
85     element(by.css('#number9')).click();
86     element(by.css('#operator_divide')).click()
87     element(by.css('#number0')).click();
88     element(by.css('#operator_equals')).click()
89     expect(running_total.getAttribute('value')).to.eventually.equal('ERROR')
90 })

```

## Screenshot of the test code passing

```
➔ js_calculator_start_point git:(master) * npm run protractor
> js_calculator_start_point@1.0.0 protractor /Users/user/codeclan_work/week_08/Unit_and_Integration_Task_8/js_calculator_start_point
> protractor conf.js

[12:03:57] I/launcher - Running 1 instances of WebDriver
[12:03:57] I/hosted - Using the selenium server at http://localhost:4444/wd/hub

- calculator functionality should have working number buttons
- calculator functionality the number buttons should update the display of the running total
- calculator functionality the arithmetical operations should update the display with the result of the operation
- calculator functionality multiple operations can be chained together
- calculator functionality is the output as expected for a negative number
- calculator functionality is the output as expected for a decimal
- calculator functionality is the output as expected for a very large number
✓ calculator functionality is the output as expected for a number divided by zero: 298ms

1 passing (576ms)
7 pending

[12:03:59] I/launcher - 0 instance(s) of WebDriver still running
[12:03:59] I/launcher - chrome #01 passed
➔ js_calculator_start_point git:(master) * █
```

## Description here

The app being tested is a calculator. The function being tested is designed to produce an error message “ERROR” when a user tries to divide a number by zero on the calculator. At first, the test fails because the error message had not been coded. I then wrote code so that the error message “ERROR” would appear. I reran the test, and the test passed.

Unit	Ref	Evidence	
I&T	I.T.1	The use of Encapsulation in a program and what it is doing.	

## Paste Screenshot here

```
ConferenceRoom.java
1  import java.util.ArrayList;
2
3  public class ConferenceRoom {
4
5      private String name;
6      private int capacity;
7      private ArrayList<Guest> collectionOfGuests;
8
9      @
10     public ConferenceRoom(String name, int capacity) {
11         this.name = name;
12         this.capacity = capacity;
13         this.collectionOfGuests = new ArrayList<Guest>();
14     }
15
16     public String getName() { return this.name; }
17
18     public int getCapacity() { return this.capacity; }
19
20     public int countCollectionOfGuests() { return this.collectionOfGuests.size(); }
21
22
23
24
25
26
```

## Description here

The ConferenceRoom class shows encapsulation. The properties name, capacity and collectionOfGuests are private. This means they cannot be directly accessed outside of the ConferenceRoom class. Instead, there are getter methods getName, getCapacity and countCollectionOfGuests that enable these properties to be indirectly accessed outside of the class. So, another class in the program can access the name only if the getter method getName is used.

Encapsulation provides a layer of security, meaning that a class can control access to its properties. This enables features such as read-only.

## Week 12

Unit	Ref	Evidence	
I&T	I.T.7	The use of Polymorphism in a program and what it is doing.	

### Paste Screenshot here

Screenshot 1

```
19     private ArrayList<Person> players;
20     private ArrayList<Enemy> enemies;
21     private ArrayList<IFight> finalists;
22
23     public Game() {
24         this.players = new ArrayList<Person>();
25         this.enemies = new ArrayList<Enemy>();
26         this.finalists = new ArrayList<IFight>();
27     }
```

Screenshot 2

```
57
58     public void addFinalist(IFight finalist) {this.finalists.add(finalist);}
59
```

Screenshot 3

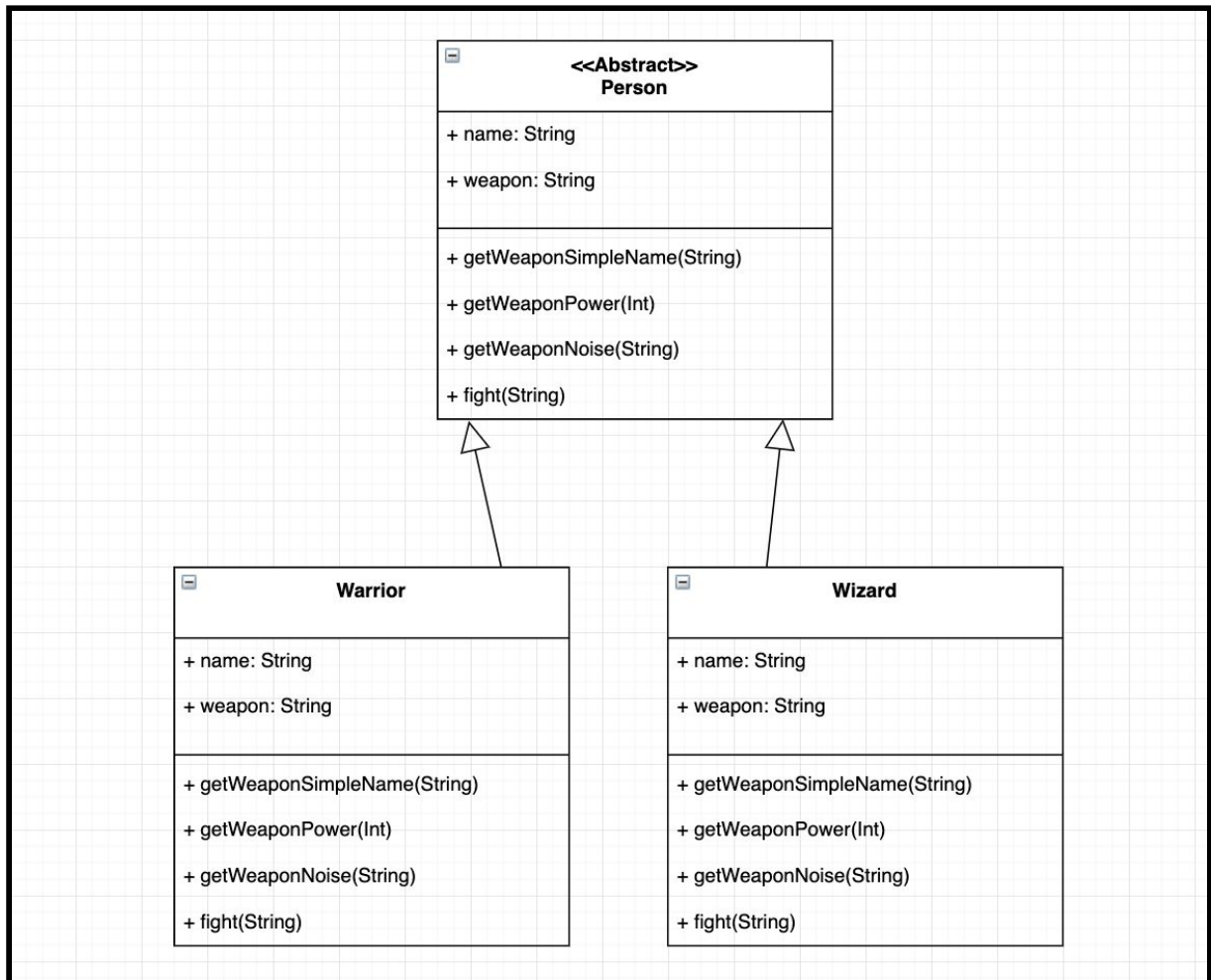
```
Game.java
102
103     public void playFinal() {
104         System.out.println();
105         System.out.println("In a dramatic fight to the death, a victor has emerged ... ");
106
107         Random randThree = new Random();
108         IFight winner = finalists.get(randThree.nextInt(finalists.size()));
109         if (winner.getClass().getName().contains("enemies")) {
110             Enemy enemy = (Enemy) winner;
111             System.out.println(enemy.getName() + " takes the prize.");
112         }
113         else {
114             Person person = (Person) winner;
115             System.out.println(person.getName() + " takes the prize.");
116         }
117     }
```

### Description here

The file Game.java shows polymorphism involving two classes, Person and Enemy. These do not share an ancestor, but both use the IFight interface. This means it was possible to create an ArrayList that could contain any object that uses the IFight interface (see the finalists ArrayList in Screenshot 1). Person and Enemy classes both use IFight, it was possible to add them to finalists using the addFinalist method (Screenshot 2). The playFinal method then selects an object from the finalists ArrayList and changes it back into its original form (Screenshot 3, line 110).

Unit	Ref	Evidence	
A&D	A.D.5	An Inheritance Diagram	

Paste Screenshot here



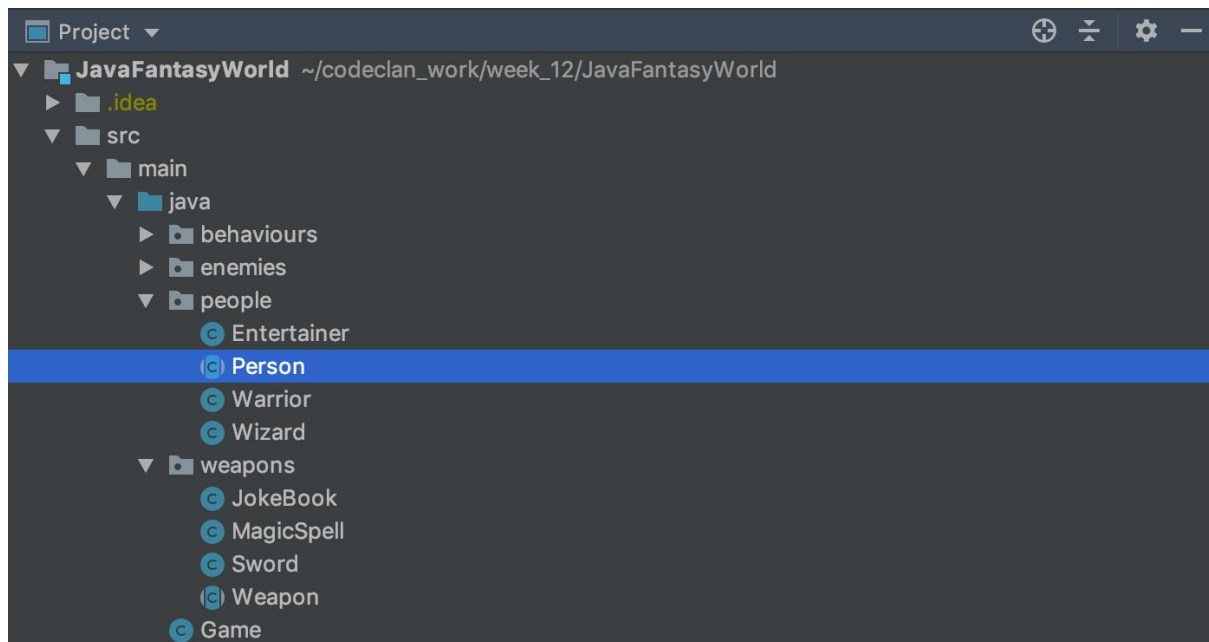
### Description here

The parent class is the Person class. It is abstract, which means that there will never be an instance of a person class; rather, each person will always be of a particular type (warrior or wizard). The warrior and wizard classes are the child classes, and they inherit all the properties and methods from the Person class. In this case, the child classes do not have any properties or methods that are not inherited from the parent.

Unit	Ref	Evidence	
I&T	I.T.2	Take a screenshot of the use of Inheritance in a program. Take screenshots of: *A Class *A Class that inherits from the previous class *An Object in the inherited class *A Method that uses the information inherited from another class.	

**Paste Screenshot here**

Screenshot 1: file structure



Screenshot 2: the superclass

```

1 package people;
2
3 import behaviours.IFight;
4 import weapons.Weapon;
5
6 public abstract class Person implements IFight {
7
8     private String name;
9     private Weapon weapon;
10
11     @Override
12     public Person(String name) {
13         this.name = name;
14         this.weapon = null;
15     }
16
17     public String getName() { return name; }
18
19     public void setWeapon(Weapon newWeapon) { this.weapon = newWeapon; }
20
21     public Weapon getWeapon() { return weapon; }
22
23     public String getWeaponSimpleName() { return this.weapon.getClass().getSimpleName(); }
24
25     public String getClassSimpleName() { return this.getClass().getSimpleName(); }
26
27     public int getWeaponPower() { return this.weapon.getPower(); }
28
29     public String getWeaponNoise() { return this.weapon.getNoise().toUpperCase(); }
30
31     public String fight() {
32         return String.format("%s attacks with a %s. %s! It has a power level of %s.", this.getName(), this.getWeaponSimpleName(),
33             this.getWeaponNoise(), this.getWeaponPower());
34     }
35 }

```

Screenshot 3: the subclass

```
Warrior.java x
1 package people;
2
3 public class Warrior extends Person {
4
5     public Warrior(String name){super(name);}
6
7 }
8
9
```

Screenshot 4: creating a class for an object to be used inside another class

```
Sword.java x
1 package weapons;
2
3 public class Sword extends Weapon {
4
5     public Sword(int power){super(power);}
6
7
8
9
10 public String getNoise(){return "Swish!";}
11
12 }
13
14
```

Screenshot 5: the object being used in a program

```
Game.java x
119 public void play() {
120     Wizard player1;
121     Warrior player2;
122     Entertainer player3;
123     EvilWizard enemy1;
124     Dragon enemy2;
125     Heckler enemy3;
126     MagicSpell weapon1;
127     Sword weapon2;
128     JokeBook weapon3;
129
130     player1 = new Wizard( name: "Thelma");
131     player2 = new Warrior( name: "Louise");
132 }
```

Screenshot 6: a method that uses the information inherited from another class



```

1 package people;
2
3 import behaviours.IFight;
4 import weapons.Weapon;
5
6 public abstract class Person implements IFight {
7     private String name;
8     private Weapon weapon;
9
10    @
11    public Person(String name) {
12        this.name = name;
13        this.weapon = null;
14    }
15
16    public String getName() { return name; }
17
18    public void setWeapon(Weapon newWeapon) { this.weapon = newWeapon; }
19
20    public Weapon getWeapon() { return weapon; }
21
22    public String getWeaponSimpleName() { return this.weapon.getClass().getSimpleName(); }
23
24    public String getClassSimpleName() { return this.getClass().getSimpleName(); }
25
26    public int getWeaponPower() { return this.weapon.getPower(); }
27
28    public String getWeaponNoise() { return this.weapon.getNoise().toUpperCase(); }
29
30    public String fight() {
31        return String.format("%s attacks with a %s. %s! It has a power level of %s.", this.getName(), this.getWeaponSimpleName(), this.getWeaponNoise(), this
32        .getWeaponPower());
33    }
34 }

```

## Description here

Screenshots 1 through 6 show the use of inheritance. Screenshot 1 shows the file structure, with a package “people” that contains an abstract class (the superclass) called Person, plus subclasses (Entertainer, Warrior and Wizard) that inherit from Person. Screenshot 2 shows the superclass Person in detail, whilst Screenshot 3 shows one of the subclasses that inherits from Person. Line 3 of Warrior is the line of code that tells the program to inherit from Person (using the the code “extends Person”). Screenshot 4 shows the creation of a separate class, Sword, which members of the Person class have. Screenshot 5 shows an instance of the Warrior class being created, and an instance of the Sword class being added to the instance of Warrior. Screenshot 6 shows several methods that are created in the superclass Person, and which are then passed down to the subclasses, including Warrior. For example, all instances of the Warrior class must have a method “fight” that returns a string. Screenshot 2 shows that the Warrior class does not override this method, so it will use the “fight” method in the format described in the method body in Person.

## Week 14

Unit	Ref	Evidence	
P	P.9	Select two algorithms you have written (NOT the group project). Take a screenshot of each and write a short statement on why you have chosen to use those algorithms.	

## Paste Screenshot here

### Screenshot 1



```

17
18 Park.prototype.findMostPopularDinosaur = function () {
19     let topDinosaur = this.collectionOfDinosaurs[0];
20     for (dinosaur of this.collectionOfDinosaurs) {
21         if (dinosaur.guestsAttractedPerDay > topDinosaur.guestsAttractedPerDay) {
22             topDinosaur = dinosaur
23         }
24     }
25     return topDinosaur
26
27 };
28

```

Screenshot 2

```

57 public void decideWinner() {
58     int playerScore = this.player.getScore();
59     int dealerScore = this.dealer.getScore();
60     if (playerScore > dealerScore) {
61         this.result = "You won! Congratulations!";
62     } else if (dealerScore > playerScore) {
63         this.result = String.format("%s is the winner. Try again.", this.dealer.getname());
64     } else {this.result = "It is a draw";}
65 }
66

```

### Description here

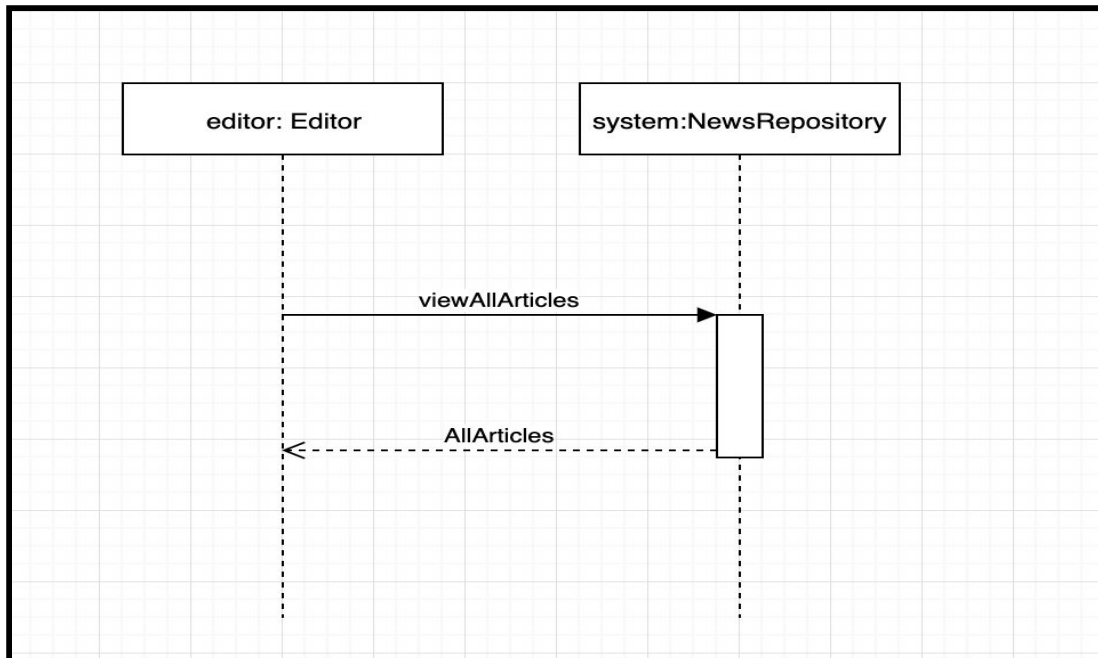
Screenshot 1: this algorithm is a function written in JavaScript. The function is called on an instance of the Park prototype. The purpose is to find the most popular dinosaur in an array of dinosaurs. To do this, we create a variable, topDinosaur, which starts as the first item in the array collectionOfDinosaurs (see line 19). A for loop is then run: the loop iterates over each item in the array. On the first iteration, the algorithm compares the first and second items with respect to the property guestsAttractedPerDay (which is an integer). Whichever item has the highest integer for this property becomes the topDinosaur. The loop is then repeated for each item in the array.

Screenshot 2: this algorithm is a function written in Java, for a version of the card game Blackjack. The purpose of the function is to determine the winner between a player and the dealer. To do this, the function gets the scores for the player and the dealer and compares the two scores. If the player's score is higher than the dealer's score, then the result is set as the string "You won! Congratulations!" (line 61). If the dealer's score is higher, then the result is set as the string "%s is the winner. Try again.", where %s represents the dealer's name (line 63). If the two scores are the same, then the result is a draw (line 64).

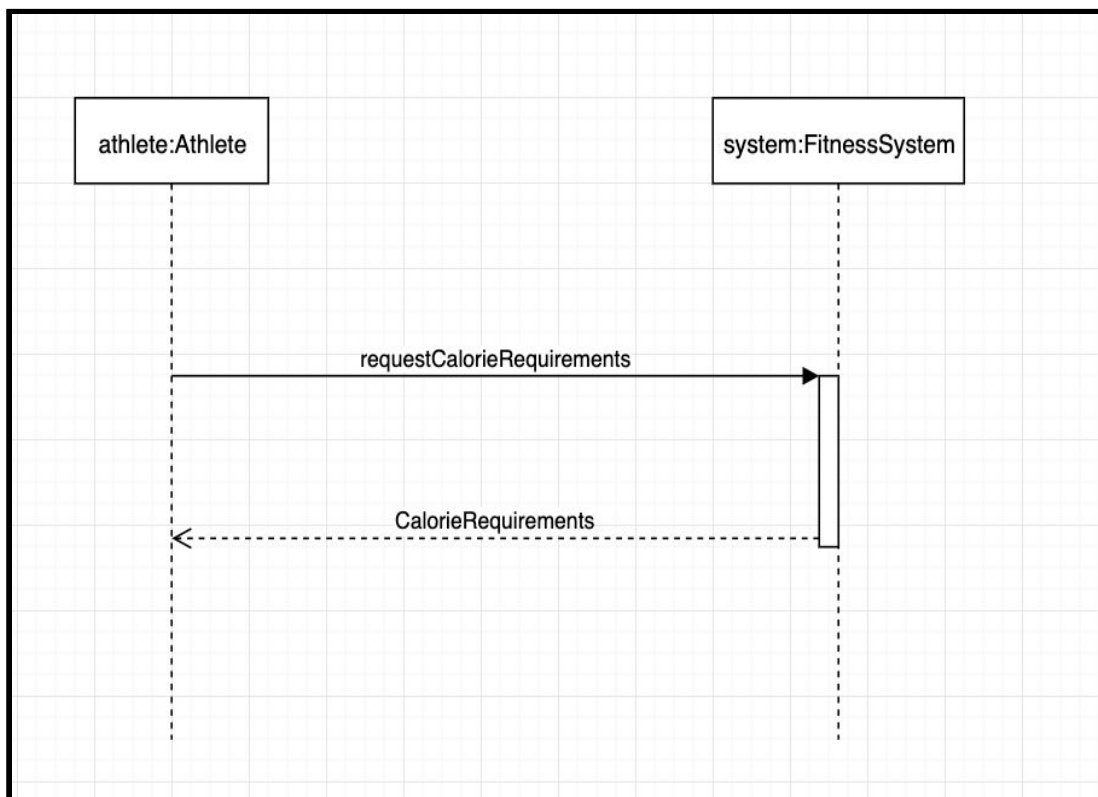
Unit	Ref	Evidence	
P	P.7	Produce two system interaction diagrams (sequence and/or collaboration diagrams).	

### Paste Screenshot here

Screenshot 1: Sequence Diagram for a News Service



Screenshot 2: Sequence Diagram for a FitnessSystem



### Description here

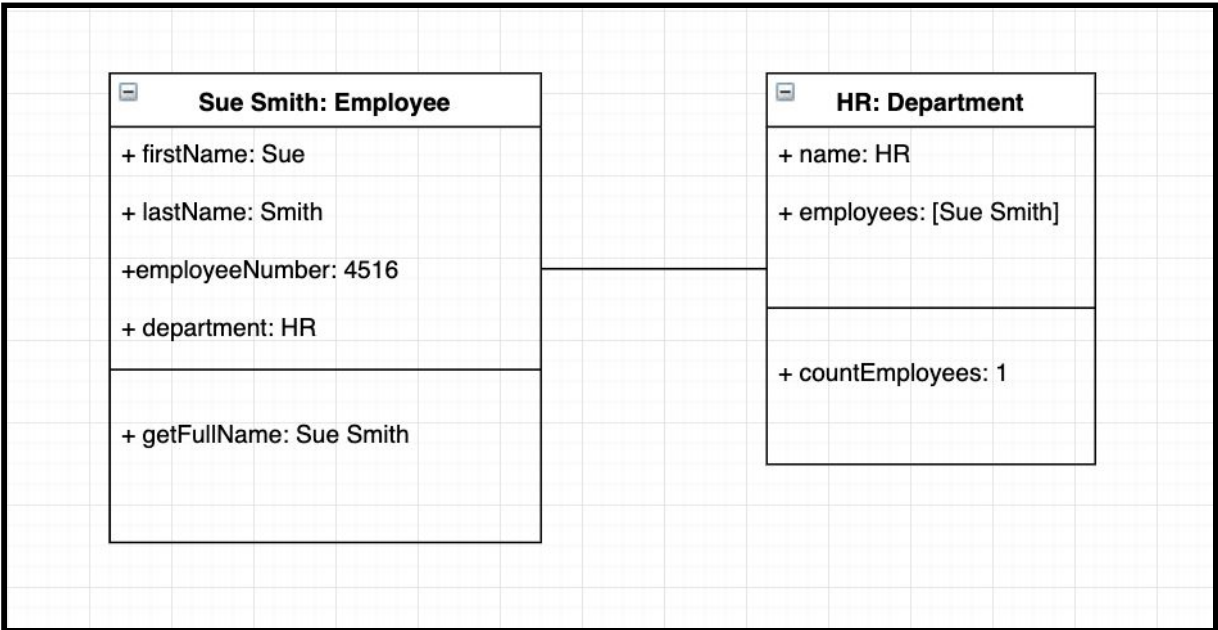
Screenshot 1 shows a sequence diagram for a news service app. An editor can send a request message to the news repository to view all articles. The news repository returns a message (the articles).

Screenshot 2: this a sequence diagram for a fitness app. An athlete requests their calorie requirements. The fitness system returns a message (the number of calories required).

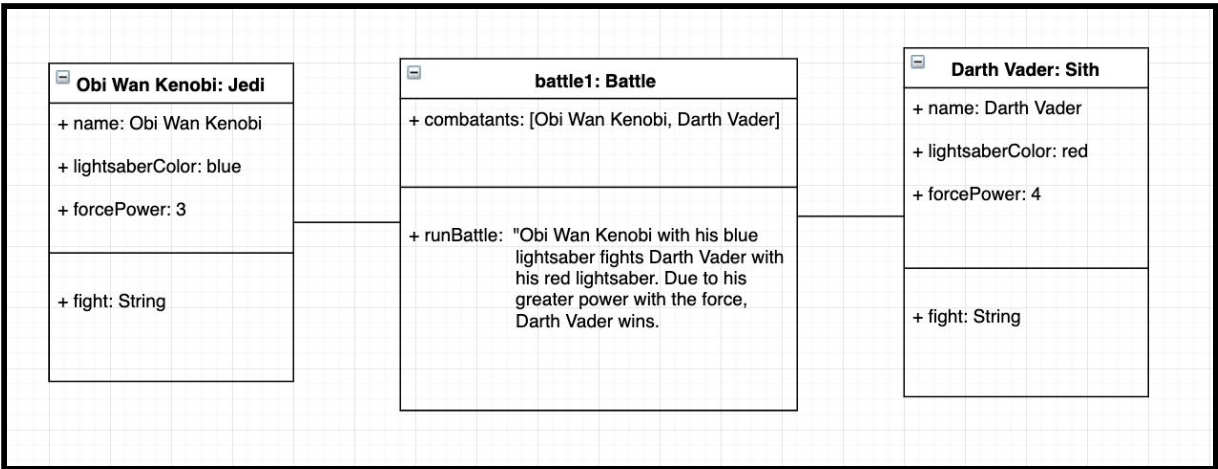
Unit	Ref	Evidence	
P	P.8	Produce two object diagrams.	

Paste Screenshot here

Screenshot 1



Screenshot 2



Description here

Screenshot 1: on the left is Sue Smith, an instance of the Employee class. The properties she has are: first name (Sue), last name (Smith), employee number (4516), and department (HR). The method `getFullName` takes the first name and last name and adds them together. The HR department is an instance of the department class. It has a name (HR) and a list of employees (in this case, Sue Smith is the only employee). There is a method `countEmployees` that counts how many items are in the list of employees and returns the result as an integer (in this case, 1).

Screenshot 2: on the left is Obi Wan Kenobi, an instance of the Jedi class. On the right is Darth Vader, an instance of the Sith class. In the centre is `battle1`, an instance of the Battle class. In `battle1`, Obi Wan Kenobi and Darth Vader fight each other. The method `runBattle` shows the result of the battle, returned as a string. Darth Vader is the winner because the method has compared the `forcePower` of each combatant and selected the one with the higher score (in this case, Darth Vader's `forcePower` of 4 beats Obi Wan Kenobi's `forcePower` of 3).

Unit	Ref	Evidence	
P	P.17	Produce a bug tracking report	

### Paste Screenshot here

Bug/Error	Solution	Date
Error when running the server in IntelliJ: Caused by: org.postgresql.util.PSQLException: ERROR: value too long for type character varying(255)	The problem was with the DataLoader: several of the instances of the story property were over the 255-character limit. We cut the characters in these instances.	5/11/2019
Radio button is not working. The "no" button does not populate, and the user cannot change their first decision.	We identified that the problem was that we had written "checked" inside the button. We removed "checked" and this solved the problem.	5/11/2019
AddArticleForm is not able to map journalists. It states that journalists is undefined.	The problem was that in NewsContainer we had accidentally passed down "journalist" instead of "journalists".	5/11/2019
DB Query to sort Articles by date descending not showing on Articles page.	Point onArticleEdit fetch to the custom search created by Spring Boot - <code>fetch('http://localhost:8080/articles/search/findArticleOrderByDateDesc')</code>	5/11/2019
ErrorPage did not run as expected. A picture should appear but instead nothing is being displayed other than an image icon	The <code>img</code> source in the ErrorPage component was not inputted correctly, fixed and now the page runs as expected.	6/11/2019
Error in rendering the "currentArticle" when we type the URL as the "currentArticle" starts null	Make a fetch for the articles using a specific id and then assign the response to the "currentArticle".	6/11/2019

### Description here

This bug tracking report shows six bugs that were solved during the project to create a full-stack news app. The first column contains a description of the bug, the second column explains the steps taken to solve the bug, whilst the third column states the date the bug was identified. Each of the bugs was solved on the date it was identified. If there had been a delay between identification and solution, a different table structure might have been appropriate (for example a column for "Date Identified" and a separate column for "Date Solved").