

Faculty of Science, Technology and Arts

Department of Computing
Project (Technical Computing)
[55-604708]
2019/20

Author:	Steven Russell Tomlinson
Student ID:	26023855
Year Submitted:	2020
Supervisor:	Andrew Bissett
Second Marker:	
Degree Course:	BEng Software Engineering
Title of Project:	A Single page music e-commerce site using a RESTful API

Confidentiality Required?**NO** ☒**YES** ☐



A SINGLE PAGE MUSIC E-COMMERCE SITE USING A RESTFUL API

Software Engineering BEng

Supervisor

Andrew Bissett

A.K.Bissett@my.shu.ac.uk

Steven Russell Tomlinson

B6023855@my.shu.ac.uk

Table of Contents

- 1) _____ Introduction
 - 1.1 - Aims & Objectives
 - 2.2 - Dissertation Outline
- 2) _____ Context Investigation
 - 3.1 - API and API types
 - 3.1.1 - SOAP vs REST API
 - 3.1.2 - SOAP vs REST Comparison Table
 - 3.2 - Multi-page applications vs Single page applications
 - 3.2.1 - Advantages & Disadvantages of Multi-page applications and Single Page Applications
- 3) _____ Similar Application
 - 4.1 - xbykygo.com
 - 4.2 - Seedlipdrinks.com
 - 4.3 - nuro.co
- 4) _____ Tools and approaches
 - 5.1 - Methodology
 - 5.2 - Repository
 - 5.3 - Front end Framework/Libraries
 - 5.4 - Front end hosting service
 - 5.5 - Back end Framework/Libraries
 - 5.6 - Database – Database tools
 - 5.7 - Endpoint API Testing
- 5) _____ Development
 - 6.1 - Requirements
 - 6.2 - Design
 - 6.3 - Front end Development
 - 6.4 - Database
 - 6.5 – Back end
 - 6.6 - Application Integration
- 6) _____ Testing
 - 7.1 - API Endpoint Testing
 - 7.2 - User testing
- 7) _____ Critical Evaluation
 - 7.1 – Evaluation of deliverable
 - 7.1.1 – Deliverable design
 - 7.1.2 – Deliverable development
 - 7.1.3 – deliverable testing
 - 7.2 – project evaluation

7.3 – issues and challenges

7.4 – ethical issues

7.5 – skill development

7.5.1 – personal skill development

7.5.2 – professional skill development

8)_____Future Development

8.1 – User advised development

8.2 – Additional development

9)_____References

10)_____Appendices

10.1 – User Testing Questions/Tasks

10.2 – Application Step Development

10.2.1 – Front End Development

10.2.2 – Database set up

10.2.3 – Back End Development

10.3 - Specification

Introduction to the report - I

This report will be trying to work out which would be more efficient and easier to use when using an online e-commerce site such as 'GAME' or 'Amazon' and why do some e-commerce sites use multi page and others use single page. Would it be better for these pages to be multi page applications or would it be more efficient for these pages to be single page applications assisted with RESTFUL API such as 'Facebook' JavaScript is used to change the display of the current screen depending on variables. The reason for this is that single page applications assisted with RESTFUL API such as 'Facebook' seem to load different pages and achieve tasks faster than that of a multi-page application such as 'Amazon' and wanted to find out if it would be beneficial for sites to start being used as single page applications now over multi page. As well as look at why some pages choose to use the type of application they do.

A solution to this would be creating a single page e-commerce music site called 'The Future of Sound' that is assisted by a back end RESTFUL API. This API would be able to send data to the front end as well as be sent data from the front end using API requests. Once this is complete, a range of people with different skill sets, of different ages will be asked to complete a series of tasks that can be completed on both my application as well as the e-commerce site 'Amazon'. After the users will be asked a series of questions about their experience to work out which the user preferred as well as which one they thought worked better by looking at things like load speed, ease of use, speed tasks could be completed and overall experience with both applications. After an evaluation would be written would the findings from the results working out which for the users was the more efficient, in terms of speed and which type of application would be more beneficial for e-commerce sites.

Aims and Objectives – I.1

This project aims to a single page application with support RESTFUL API against a secured multi-page application such as Amazon to work out which would be most beneficial for an e-commerce site to use to complete it's business and which one would be easier and more efficient for the user to use.

This also aims to identify what a RESTFUL API is and how it can be used with single page applications to increase the speed and efficiency of a single page application in order to properly evaluate it against a large multi-page application.

Finally this project aims to develop an understanding of what the difference between a multi-page application and a single page application and how a single page application can be developed such as 'Facebook' that is able to be supported by the created RESTFUL API stated in the above paragraph.

In order to achieve the aims above the following objectives are defined:

- Create a front end single page application using a supported framework such as React.js or Angular
- Create a back end RESTFUL API application to support the front end that is able to send data from the database connected to the front end application for products as well as being able to send data from the front end back to the database using the API using a framework such as Django or express.js
- Asked a range of people at different ages and skill levels to complete a series of simple tasks and answer a series of questions about it after using both the single page and multi-page application
- Evaluate the results from the users experience of both sites to work out which is better for the user and which may be a more efficient and easier to use application overall.

Dissertation Outline – 1.2

As a basic outline for this dissertation a simple single page application will be built using React.js JavaScript framework to develop a front end design for the user to navigate through that will be hosted on a local server using node.js. This will include manipulating data from the API and displayed in a consistent format as well as adding the capabilities to send data from it to the database using request. A search bar will also be added in for the usability of an e-commerce site so users can locate specific information.

After this part is complete the back end of the application will be built using the python framework Django to create a RESTFUL API that is able to handle requests coming from the front end application that will provide either the front end or the database with data. This will include such functions as getting all data, getting data using keywords from a search bar in a filter, sorted data using categories as well as the ability to edit and delete information using API requests. Such data items will include products for the site, purchases made on the site and handling information in the user's basket.

A database will also be set up from this using a Mongoose Database to store all the information needed for this project to complete the functionality specified above so a working e-commerce site can be accurately built.

At the end of development users will be asked to complete a series of tasks on both applications (multi-page and single page) and then asked a few questions about the usability, easy of use, general appeal, speed in which tasks could be completed and general speed of the web pages. Asking users, the questions will also be a part of user testing for the application. This information will then be recorded anonymously and added to a display format before being starting to evaluate a multi-page e-commerce site to a single page e-commerce site.

Context Investigation – 2

API and API types – 2.1

To best understand what a RESTFUL API is first you must be able to understand what an API is.

An API or Application Programming interface is a set of functions and operations that have been pre-set so that an application is able to access features and sources of data from another external system(API Definition, Lexico.com, 2020). 'Think of an API like a menu in a restaurant. The menu provides a list of dishes you can order, along with a description of each dish' (what is an api, howtogeek.com, Chris Hoffman, 2018). If you think of an API like Chris did in the previous quote you can specify what you want from the menu, using a API request, and the kitchen (external system) does all the work collecting the ingredients and sending it to you. This is how an API works. You send it a request for data you need from another system and it will collect all that information and send it back to you. You don't need to know how it does it. Only that it's sending you the correct data or function you asked for.

Not only can the user request data to be brought to them but they can request for data to be sent to the external system. For example, if you were updating some user information on an application such as 'Facebook' you would be sending a request to the 'Facebook' API to update your user credentials. This way the next time you visit your account your new information will be displayed when the page requests your information.

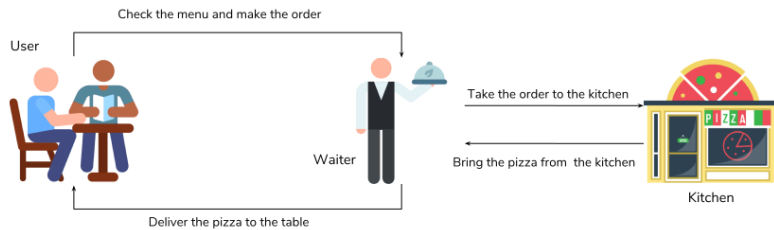
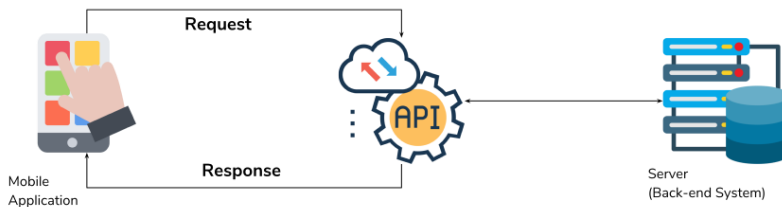


Fig 1 – How API Works

An API can save a developer a lot of time as they will not have to perform all the messy back end development if they are able to pull it from an external source. Reducing the amount of code they would have to produce to complete a task as well as creating more consistent apps if all the data being sent and received is all in the same style and format

Information gathered from (Medium, Amanda Kothalawala, 2018, What is an API? How does it work?), (Lexico, Oxford, API),(HowToGeek, Chrishoffman, 2018, What is API)

SOAP vs REST API – 2.1.1

The most used types of API for sending and receiving data are SOAP API and REST API. These API both complete very similar tasks when used like an API but both have different benefits and drawbacks. So here we will be comparing the two to see which could be the better API for an e-commerce application.

SOAP or Simple Object Access Protocol is the first type of API that could be used for an e-commerce site. SOAP is a messaging protocol for constantly changing data in a decentralized and distributed environment. The difference between these 2 are decentralized has a centre that everything connects to but has a subset of systems that have links to them too. So, all the subsets have systems that connect to them and the subsets connect to the centre, a little bit like a tree with its branches. Every point in this tree has its own point where decisions are made instead of it all coming from one place.

Distributed on the other hand means that processes from these systems are shared between many interconnected nodes at once. Like synapses in the brain but still centralized and use the complete system knowledge instead of just those nodes it's connected to.

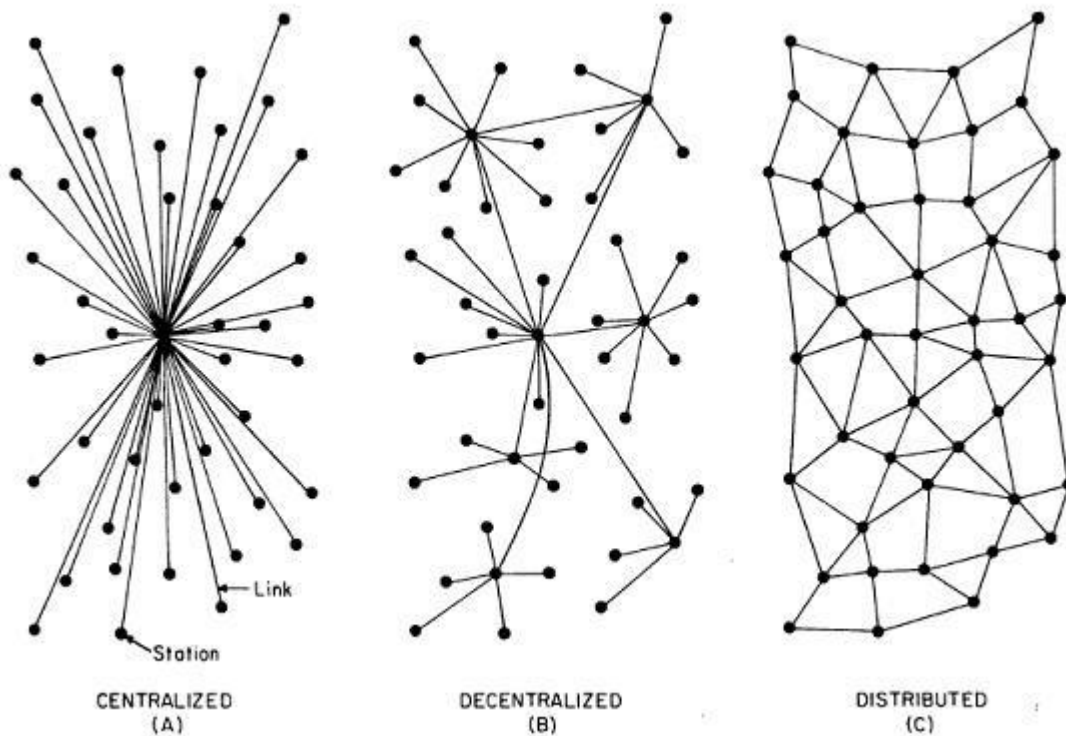


Fig 2

Not only this but SOAP allows for the API to along side many different layer protocols such as HTTP, SMTP, TCP and UDP and is able to output any data sent from it in the standard XML format including the standard envelope, header body and fault layout like so.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Header>
    <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
      <n:priority>1</n:priority>
      <n:expires>2001-06-22T14:00:00-05:00</n:expires>
    </n:alertcontrol>
  </env:Header>
  <env:Body>
    <m:alert xmlns:m="http://example.org/alert">
      <m:msg>Pick up Mary at school at 2pm</m:msg>
    </m:alert>
  </env:Body>
</env:Envelope>
```

Fig 3

Such things as error handling, security and authorization are all built into the SOAP protocol whereas REST doesn't. Something SOAP doesn't need where REST does is point to point communication which is used for synchronous communications. Meaning SOAP can complete multiple tasks all at once whereas REST would need to complete each task individually using a point for each request. So, SOAP would be great in a large enterprise application where multiple tasks can be completed through different systems and can all be done with one call to the API. 'SOAP also follows a standardized approach that clearly tells the application how to encode the XML files sent back from the API request'(soap vs rest, raygun.com, Anna Monus, 13th Feb 2020) This makes data handling very consistent and straight forward.

This type of API would primarily be used for enterprise level applications that require a higher level of security and complex transactions that require a little more time to complete such as the PayPal public API. This API allows users to log into e-commerce sites using there PayPal account to complete transactions on that site as well as allowing the owner to add there PayPal details to their site so transactions can be directly added to their account. This API can do a wider variety of actions, but these are some of the most common. Large scale sites that require Payment gateways as well as identity management would commonly use SOAP for its built-in security features.

REST or Representational State Transfer is the second API choice that could be used with an e-commerce site. REST as far as APIs go are a lot more flexible and customisable than SOAP APIs but at the same time limited in certain ways compared to the SOAP API. Rest is an architecture style that sets up a set of recommendations for creating loosely coupled applications that use HTTP protocol, just like SOAP, to send and receive data from the database. Although unlike SOAP, REST doesn't have say how to use it. SOAP has a set of rules set in place so that the application knows how to handle incoming XML files returned by the API. But because REST doesn't have any of this means that developers for this API can define how it works specifically for their system. They can define what security to implement, what format the data is returned in be it HTML, JSON or XML and is able to fine tune this API to do what ever they need it to be able to do at any stage. Web services using REST API are called RESTful web services.

```
{
  "username" : "my_username",
  "password" : "my_password",
  "validation-factors" : {
    "validationFactors" : [
      {
        "name" : "remote_address",
        "value" : "127.0.0.1"
      }
    ]
  }
}
```

Fig 4

'REST was created to address the problems of SOAP'(Soap vs rest, Raygun.com, Anna Monus, 2020).

REST is popular choice among developers to build APIs because of how much more flexible they are to develop. Once you developed how it works and how it sends and receives data then this API can be accessed through a series of functions throughout any application that can access this data. For example, when building an e-commerce site, you would have a function for adding data to a purchases database that keeps a record of all purchases on that site. This part of the API could then be accessed by a different application for administration so that they can see all the data in the database and would be able to still use the same API commands and receive it in the same or a different file format. Thee type of API are better for smaller systems such as stand-alone e-commerce sites and small internal systems where not many functions have to be completed at one time.

The main components of a SPA include the following:

- Javascript libraries and frameworks; Using an MVC framework and JS libraries
 - Routing; something to manage all the different views available on the SPA
 - Template engine; used to render or rerender parts of the SPA on the clients side of the app
 - HTML5; this is used to create the main web pages for the application and general design. Javascript will be used a lot for the APIs communication
 - Back end RESTful API; The server acts as a web API that the app is using and doesn't include a server side rendering option for web pages
 - Ajax; All communication with the server is asynchronous calls using Ajax. When a response arrives, the web page is partially rendered using js (if needed) it will re render the new information for the page rendered
- (Pro Single Page Application Development, Gil Fink, Ido Flatow, SELA Group, 2014)

'According to Nordic API's, REST is almost always better for web-based APIs, as it makes data available as resources, like user, instead of a service, like getUser.' – Anna Monus, SOAP vs REST 2020

Another main part to REST is that REST inherits HTTP operations so that when the developer is working on how to send or receive information they can use the standard well-known to developers, HTTP verbs to complete specific tasks such as GET, PUT and DELETE. Or to a none developer GET the information, which can be used to create new data also, PUT, which is more often than not used to edit data and DELETE which like it says on the tin, Deletes information from the database using the API

SOAP vs REST Comparison Table – 2.1.2

Feature to compare	SOAP	REST
Advantages	High Security, standardized data, easily extended	Scalability, much faster, browser friendly, very flexible and customisable
Disadvantages	Poor performance compare to REST, a lot more complex, not as flexible	Not as easily secured, not suitable for large scale operations, only supports HTTP protocol
Transfer Protocol	HTTP, SMTP, UDP and more	HTTP Only
Message Format	Only XML	Plain text, HTML, XML, JSON, YAML and more
Performance	Requires higher bandwidth and computing power	Requires less processing power and fewer resources
Security	WS-Security with SSL support. Built-in ACID compliance.	Supports HTTPS and SSL
Approach	Function Driven (uses functions like getUsers like using classes)	Data-driven (data available as resource like 'User')

SN	SOAP		REST	
	Size of requested data(in Bytes)	Response time (in ms)	Size of requested data(in Bytes)	Response time (in ms)
1	422	500	54	121
2	424	521	50	118
3	423	520	49	115
4	425	528	49	115

Fig 5

(Semanticsholar, Anil Dudhe, Swati S. Sherekar, 2014, Performance Analysis of SOAP and RESTful Mobile Web Services in Cloud Environment)

Due to looking for a faster more efficient application to analyse its effectiveness and efficiency to an enterprise level multi-page application, a REST API will be used when developing the application for it's speed, it's ease of customisability as well as how many different formats of data the API can handle. This will also allow me to use the standard HTML verbs when getting or sending data either on the application itself or when testing the application.

Information gathered from (Semanticsholar, Anil Dudhe, Swati S. Sherekar, 2014, Performance Analysis of SOAP and RESTful Mobile Web Services in Cloud Environment),

(Medium, Mari Eagar, 2017, What is the difference between decentralized and distributed systems?),

(Raygun, Anna Monus, 2020, SOAP vs REST vs JSON – a 2020 comparison),

Multi-page applications vs Single page applications – 2.2

What is a Multi-page application? A Multi-page application is your more basic web applications that will load an entire page of content, styling, data and images every time a link in said web application is pressed. A multi-page application consists of multiple files all containing requests to a database, if needed, and multiple different sets of HTML code that has to be run and loaded every time a page has loaded. This process does take time as the data has to be found first before it can start to generate the page the data is for. The page must render on the browser, send the data and page to the client and display it on the browser. This can take even longer if data from a large database has to be rendered in the page selected too.

‘Multi-page apps can also render specific components to the page using AJAX’ (Single page vs multi-page apps, Goldy Benedict Macquin, Medium.com, 2018). Instead as AJAX can be activated using a button or key press while on the page to load different or new information on the screen which would make the application faster. But, this would make the process even more difficult and complex as the ajax will be contained in a different file that needs to be called once the page has finished loading and requires a connection and a lot of JavaScript to complete this task.



Fig 6

What is a Single page application? A single page application, unlike multi-page apps, only use 1 HTML file for the whole application where every different page will be rendered from. So instead of have multiple files with similar HTML and CSS on all the files that have to keep re-rendering the pages, single page will render constants, like the header for the page and the background, one time then will never have to do it again. Instead it will change what is on screen by what the URL has in it. Then the application will change the pages content using a Script of some sort, a little like how AJAX works, so it seems like you’re loading different pages but it’s all being run from one page.

Single Page applications display information in a simple, elegant and efficient way due to it being able to load the whole application in one go instead and one file instead of having to navigate to different pages and render them again. ‘These types of applications are much faster because they complete all the logic for the app on the browser instead of server side’ (Single page vs multi-page apps, Goldy Benedict Macquin, Medium.com, 2018). So very little server work is needed making the page loads much faster. The only time a SPA (Single Page Application) will need to access a server to get anything after the initial load would be to collect data such as product information, user information and other sets of data that can change through time.

Advantages & Disadvantages of Multi-page applications and Single Page Applications – 2.2.1

	Multi-page Apps	Single Page Apps
Advantages	Works well on a search engine, Visual map of where to go is a key part of all multi-pages, easy to perform correct SEO, easily edited to expand the application	Run smooth, Faster, easily developed, easier to Debug with it all being on the browser, easy to convert to a mobile app. Can use the same back end code. Easy to cache data, all-round easier to use
Disadvantages	Slow in speed and performance, takes longer to develop, requires	Slow for initial load as everything is loaded at once, tricky to set up

	a lot of maintenance and updates especially with security.	Asynchronous data exchange, requires javascript to be available and enabled on the browser, less secure to multi-page apps, risk of memory leak
--	--	---

(single page app vs multi page app, rybygarage.org, Anastasia Z, 2018)

(single page application vs multi page application, Medium.com, Neoteric, 2016)

After looking at both types of web application as well as their advantages and disadvantages as well as taking into account what project aims and objectives, a single page application will be used along with the REST API. These two both had the advantages of being easier to develop as well as being both fast and efficient when being used. It also makes things easier when it comes to testing as single page applications are heavily browser based so being able to use the chrome developer tools as well as browsers developer tools. Also, for future reference it could also be easily converted to mobile app if I needed it to be. But the main reason is the fact that, SPA's are smoother, faster and all round a better user experience for clients.

Information gathered from (Pro Single Page Applications Development, Gil Fink, Ido Flatow, SELA Group, 2014, Using Backbone.js and ASP.net),

(Medium, Neoteric, 2016, Single-page applications vs Multiple-page applications),

(Medium, Goldy Benedict Macquin, 2018, Single Page applications vs multi page applications – Do you really need an SPA?),

(rubygarage, Anastasia Z., 2018, PWhat's the difference between single-page and multi-page apps)

Similar applications – 3

To make sure that the design and development of the application is up to standard it is vital that first other SPA are taken into account. This way we can find the key components of each one, analyse how they work and hopefully understand the benefits and drawbacks of each one as well as SPAs in general to make sure the application in development meets the user's expectations.

Multiple pages have been found through this research of other SPAs all with UI design, functionalities and it's over all ease of use so it would be unfair to compare the application being developed from one example, so 3 have been selected to collect a broad range of information about different types of e-commerce SPA and hopefully bring components of each one into the application being developed not only to make sure standards are met but to make sure aims and objectives are completed too.

All examples provided we located on onepagelove.com(<https://onepagelove.com/inspiration/e-commerce>)

Example 1 – 3.1 – xbykygo.com

Xbykygo is a standard single page e-commerce site that sells noise-cancelling headphones of a number of varieties. At first glance you are able to see that this is not a large scale enterprise level application of business like Amazon as they only have a small amount of stock and the products aren't very well known through out pop culture due to them having a number of well known brands that are huge competitors such as beats. But, it's still a very clean and professional looking site

Fig 7

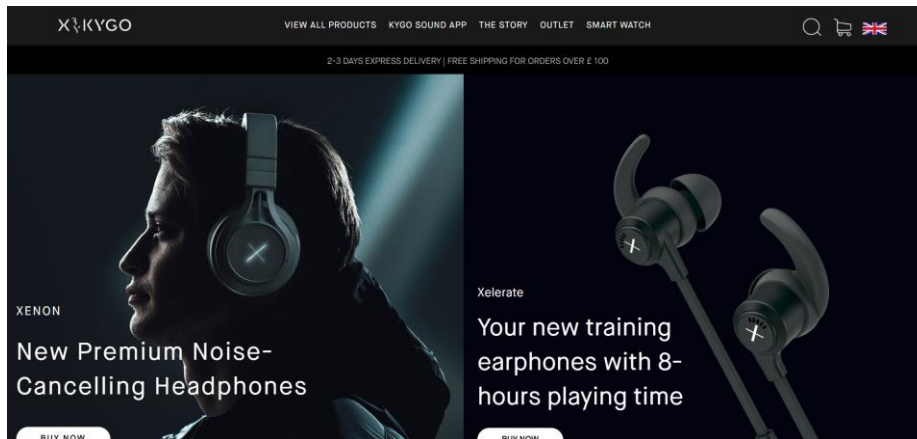
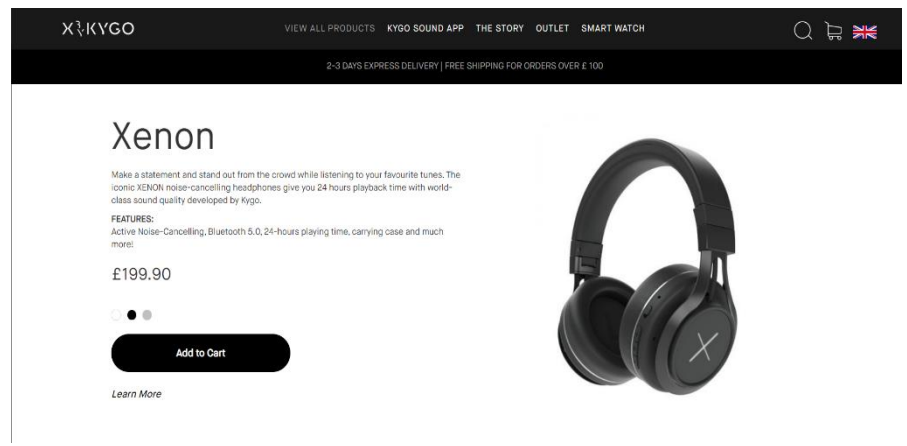


fig 12

xbykygo.com Web Design – 3.1.1

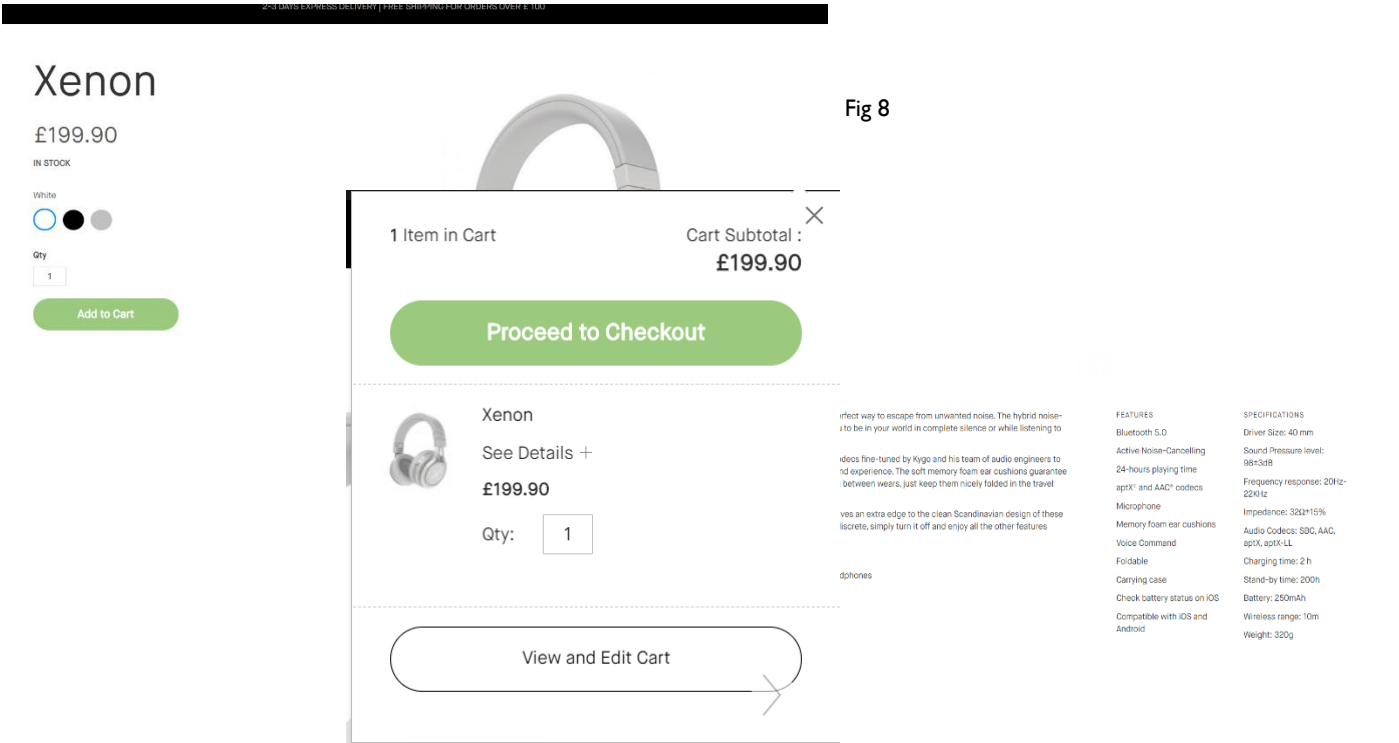
The general design for this application is quite straight forward and easy to understand. It starts out with a Black and white header like most e-commerce sites so the user is able to see paths to other parts of the application such as products, the app, The story etc making it very clear what does what and what link sends you where. It also has a search bar built into the header along with a link to the basket and link to choose the language you would like to view the site in. Keeping all these key links in the header makes general use of the application a lot easier than if they were scattered through several pages. This way the individual pages only show what they need to. No more and no less. The contrast of the colours also makes everything in the header clear to see for all users, even the logo in the top left. This is due to the products colour scheme being black and white but also to make sure it's easy to read.

The colour scheme is constant throughout the whole application as it can be seen on every page. Keeping things constant is a good part of the application so that the user doesn't get confused thinking they have entered a different site if the colours randomly change colour. Plus the brand has to keep face with their colour scheme so changing that may make people believe they aren't as professional.

The design of the item pages are really tidy. Names, prices, quantities and colours all clearly displayed on one side clear and in large lettering so it's clear how to use it, what the price is and how to select a colour you want and a large button clear as day to clearly show how to add the item to your cart. Below this is the product information. Again, all information needed is all clearly displayed and easy to read. The font is basic and understandable, the lettering is the good size but maybe a little small for those visually impaired but the general design and layout of it all is clean cut and easy to understand. The about page is laid on the

left about the product generally and what the product includes, and on the right you have the more technical knowledge for the more tech forward user.

Finally the checkout page and basket. The basket clearly displays what you ordered, how many you ordered as well as how much that item and over order costs before your eyes reach the checkout button



below. This was not a separate page. Basket was a modal displayed when the basket icon was pressed in the top right corner. Once check out is pressed it takes you to the checkout page. This is a nice simple column of input boxes clearly displaying with a "*" symbol what is required of the user. It clearly identifies what details need to go into each box and has split up all the different components nicely so that it keeps the overall look of this page clean instead of crammed together.

Fig 10

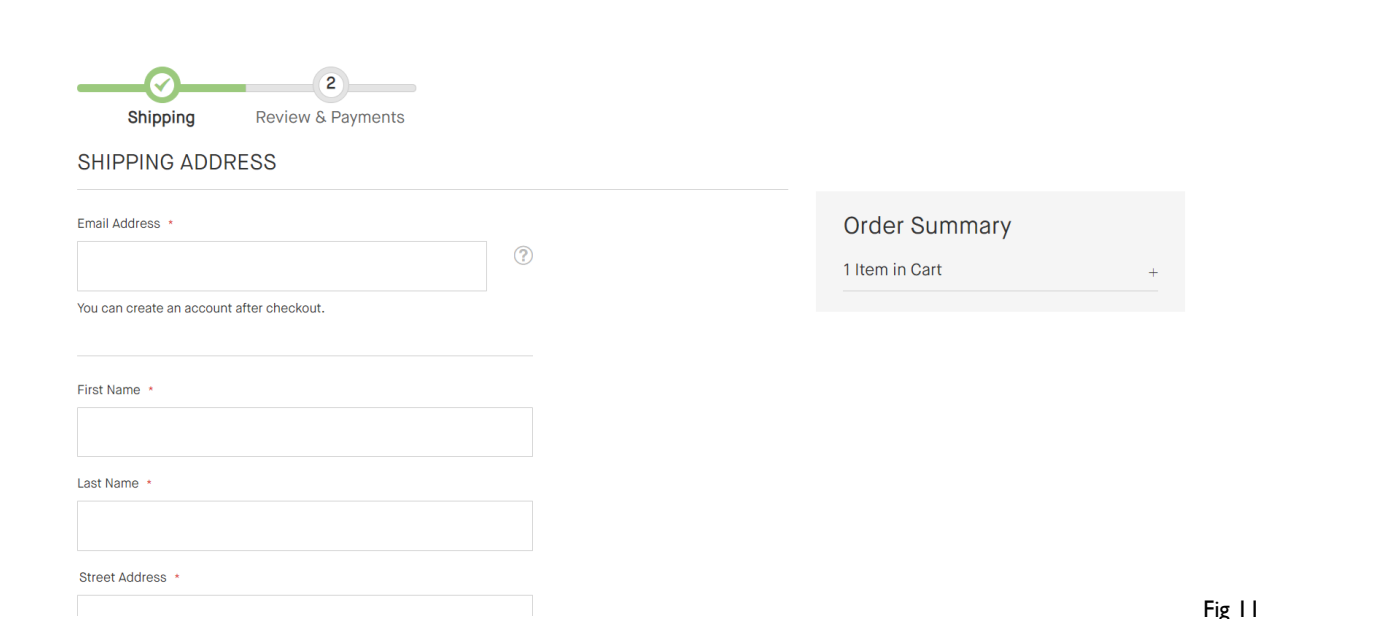


Fig 11

Overall this SPA has a very clean and easy to understand UI that makes the site and the company look very professional while also keeping it simple enough that any user that does use it will be able to pick up how to complete specific tasks very quickly and easily. Giving them an all-round pleasant experience with the web page.

xbykygo.com Functionality – 3.1.2

The functionality for this SPA is straight forward as far as e-commerce sites go. It has the absolute necessities it needs to be able to complete the task it was developed for. It allows users to search for items, click on items they want to purchase, choose a colour and quantity for the item, view the item in the basket and purchase the item through the check out option. It also allows for quick as easy navigation through the webpage using the links provided in the header at the top of the screen if the user was looking for something specific on the site.

When the site has loaded the user gets to scroll down the page to look through the companies products or navigate to another page using the links at the top of the screen. Once the user has chose a product they are able to press on the image of the link under the product to go to another page that tells you a little more about this product. From the product screen the user can select a colour for the product. The image of the product will then change to the corresponding colour. They are also able to choose from a dropdown box how many of this product they would like. Once selected the user is prompted to select add to card. This will then start a small loading animation before confirming the item has been added to the basket. Also, on this screen is a set of buttons either side of the image. This lets the user switch between angles of the product so they can have a good feel what the product will look like upon delivery.

Once the user has an item in their basket, they are able to click the basket button in the top right to view all the items in their basket. From here they can view their purchase details as well as edit their order. From here the user can also select to check out. This will take them to a standard delivery details page where information such as personal info, address and card details will be taken before pressing purchase. Once purchase is pressed a confirmation page will display if it was a success or not then take you back to your order or back to the home page

Site used for app research: (xbykygo, 2020)

Example 2 – 3.2 – Seedlipdrinks.com

Speedlip Drinks is another example of a single page e-commerce application. This application was created to sell and distribute 'The Worlds first distilled non-alcoholic spirits. Not only do they sell spirits but mixology equipment too such as measuring glasses for shots, bottle openers, cocktails and much more while promoting an option for people who aren't or can't drink

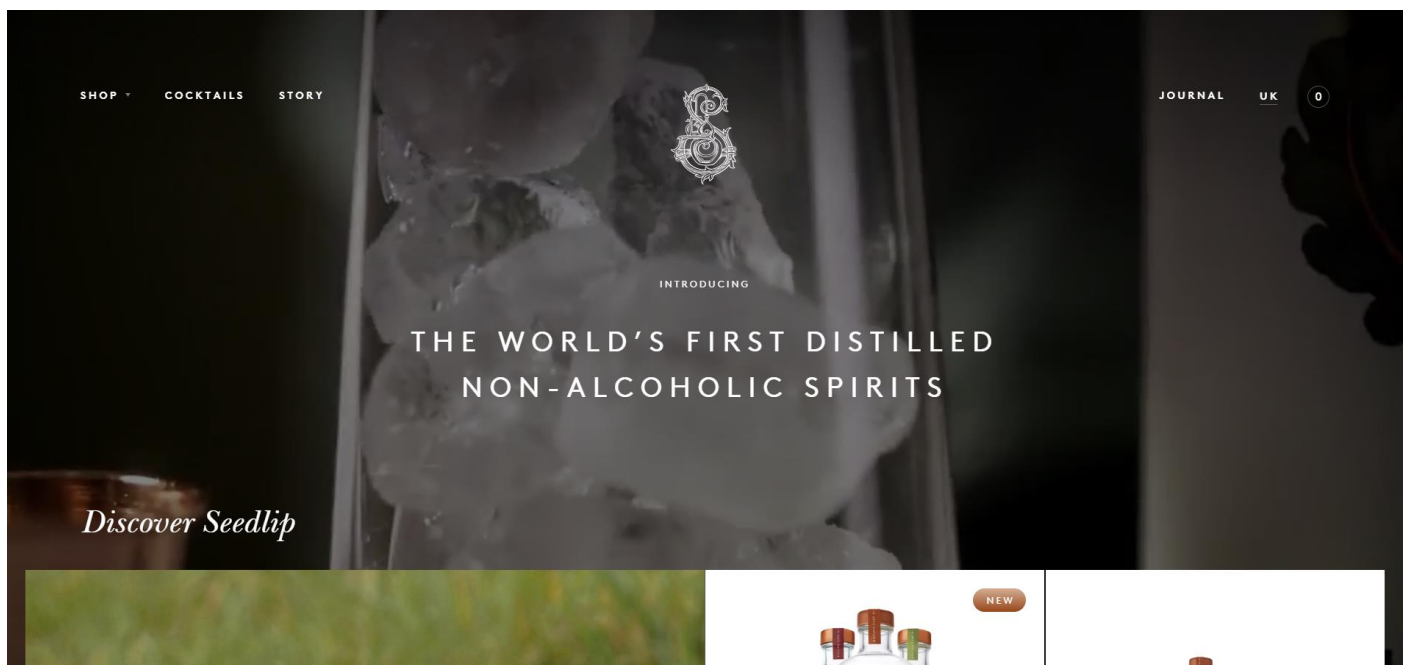
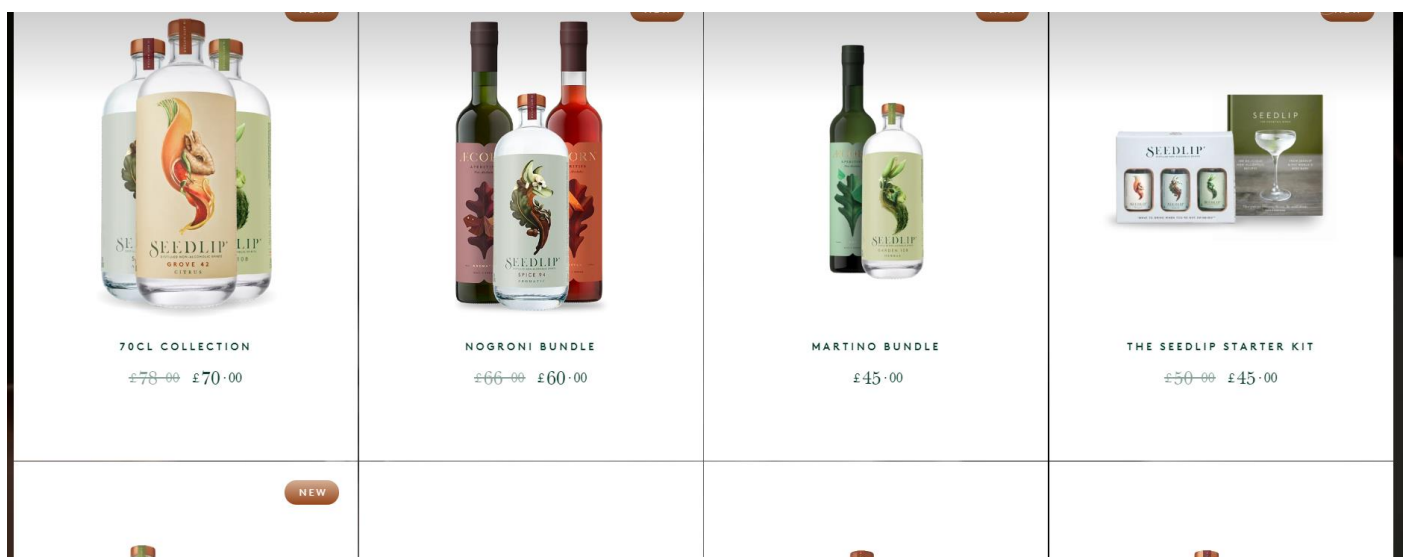


fig 13

Seedlipdrinks.com Web Design – 3.2.1

The design for this application is not unlike the previous application with there being links in the header to clearly display to users how to navigate between pages. With the design you start out with this head as seen above with an animated picture for the background clearly showing how the product is made which also has over the top of it the logo for the company and what the company is known for making it clear to the user. Again, all the text contrasts well with the background so the text is easily visible and readable to the target audience. All the colours used on this application are very vibrant and warm. This will be because of the product they sell. They are different types of non-alcoholic spirits and cocktails and there packaging has been made to look warm and appealing to the user in the hopes it will allow them to sell more of that product. The animated background is also very eye catching due to its animated nature. It gives the page a modern twist on a very retro looking logo.



All the products on this page that are found in the shop link are all display in a list like format where each product gets it's own card of sort which has an image of the product and it's price, sometimes it's reduced price too. Once a card is pressed the user is then met with a similar style to the last application. All the information is set in a nice cream box on the left with a good contrast on the font colour and the product in question is on the right. Below is even a video to show the user how that specific drink is made. All in all it's a very elegant style for a very elegant collection of high quality spirits with just enough of a modern twist to appeal to the older demographic as a classic and the younger demographic as modern and new. Each page even comes with a large bold title, so you always know what page you're on and the links to the other pages are shadowed out when you're already on that page

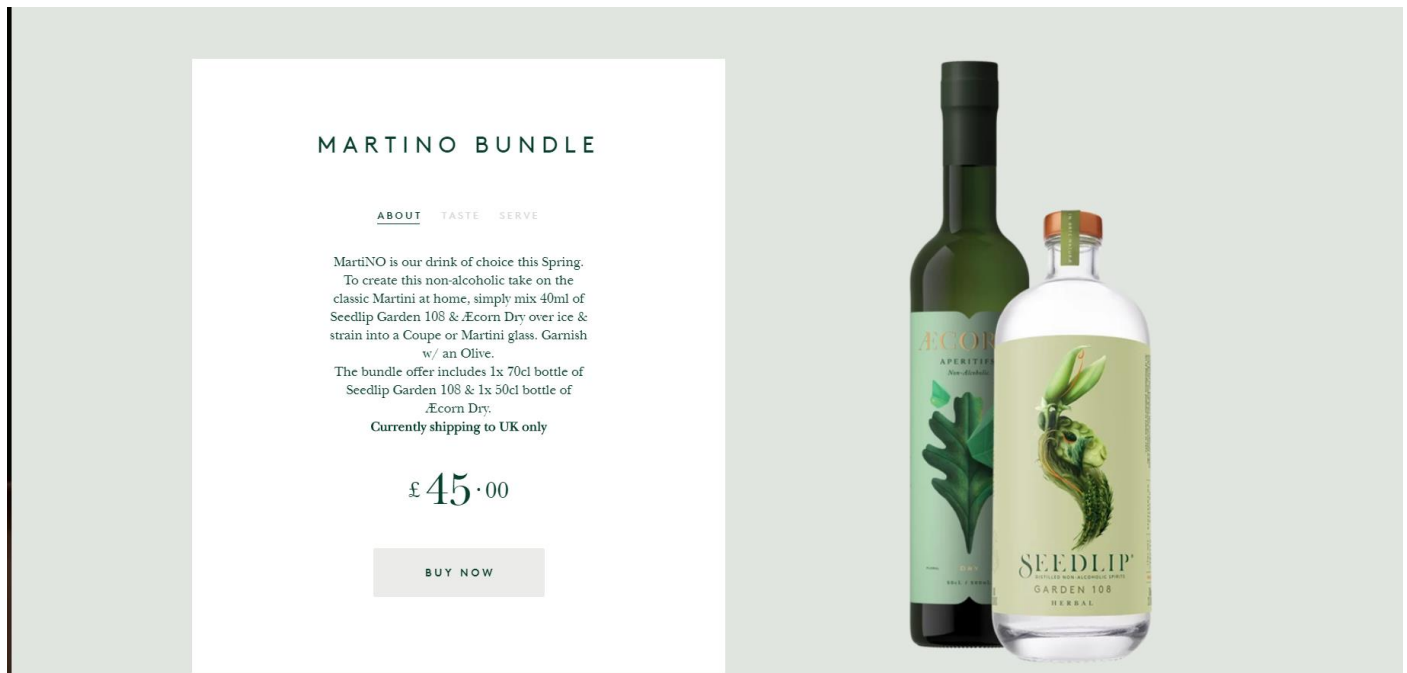


Fig 15

Seedlipdrinks.com Functionality – 3.2.2

The functionality to this is no unlike the previous example. With how the general operation of the website works. Just like the previous example all the bottom at the top in the header will help the user navigate through the web page with ease as they are all clear about what they do as well as how the buying of the product works too. The background of the application does work differently as this time the header gets smaller as you scroll down and contains an animated picture instead of just a black box. This way the header always stays at the top of the screen, so the user won't have to keep scrolling back to the top to press a different link

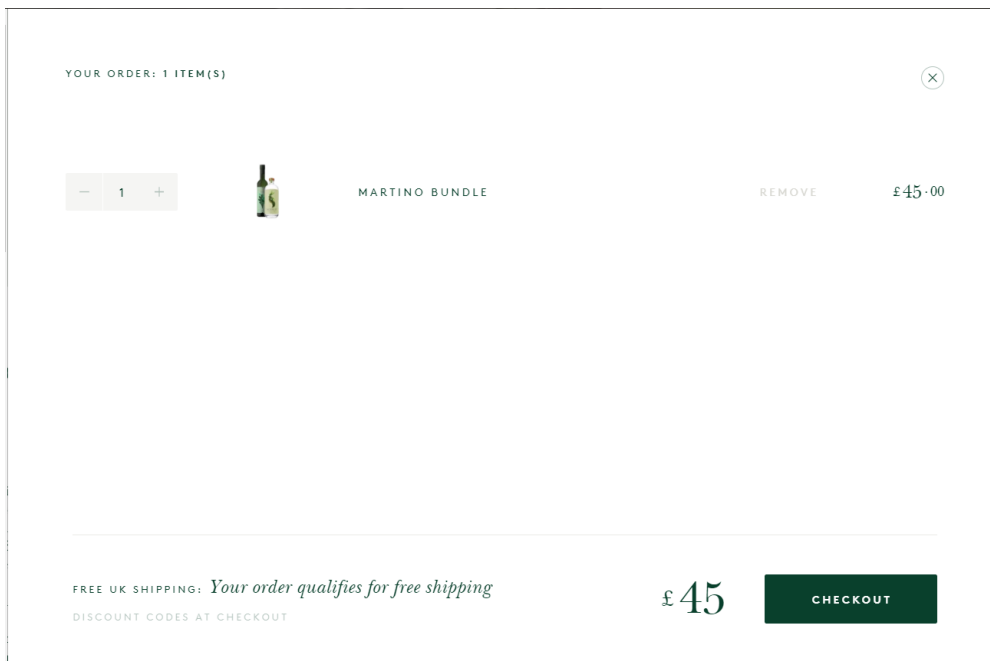


Fig 16

Once you press on an item on the shop menu it will take you to a page where the item is displayed along with the price. From here you can click to add the item to your basket. This time once you press the 'buy now' button it will open up a modal from the right-hand side of the screen instead. This will be the user's basket. From here the user can remove items from their basket, update the quantity of an item or proceed to check out instead. On check out also, this application has a few extra options to the previous app. This application has brought in external API for checking out and paying using a much faster method. They have introduced the use of external payment methods such as ShopPay, Google Pay and Amazon pay. This way the user can press one of these links, log in with their details and avoid having to input all the delivery and card information for the order. It is all provided through this 3rd party app. They have also provided an opportunity to add a discount code next to this express check out option. Allowing users with specific codes to get discounts off of their purchases. The rest of the checkout page is very similar in how it works to the previous example, just with a different styling. You can see the example for the basket in fig 16 and the checkout with express option in fig 17

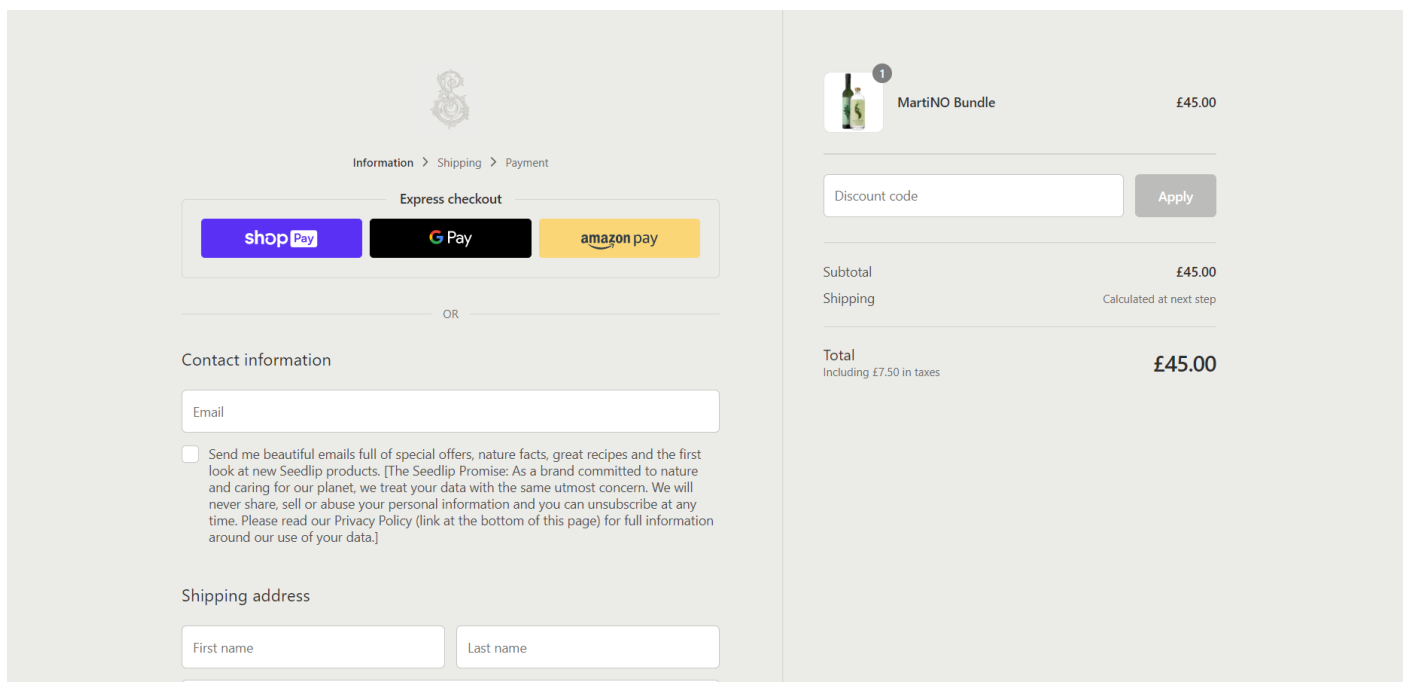


Fig 17

Site used for app research: (Seedlipdrinks, 2018)

Example 3 – 3.3 – nuro.co

Nuro.co is an online retailer that sells the NU:RO Brand watch and this product only. This just like the other 2 examples is not a large-scale enterprise but instead is a local business that has turned to an e-commerce site to reach a larger audience with their product. This like the previous 2 examples this application is again a SPA and so will provide useful when working on the design of the project at hand.



fig 18

Nuro.co Web Design – 3.3.1

This SPA is a little different to the previous 2 applications with how it clear it is to the user to use, how products are displayed in a much more concise way and the general design over all such as how the web page is laid out.

First of all, there is a clear difference between the first 2 applications and the current one. NU:RO doesn't have a header or links at the top going to separate pages with products and information on said products. The only link provided that is near where a header would be is at the top of the screen labelled 'Buy Now'. The home page as you can see in fig 18 does resemble that of example 2 as it contains 1 large image of the product or something related to the product covering the background with big bold text on it, that contrasts well with the background colour, that sends a clear image to the user. It is pretty clear from the first look that the web page is targeted at watch collectors or people who like a different style watch to the traditional one. The background image is also animated which will help catch the eye of the user as the dials on the watch do rotate like a full functional watch as well as the strap on the back of the watch changing colour, clearly marketing the varieties of colour the product comes in. This page looks quite modern too so, it is expected to reach more towards the younger generation/business person. The colours like on example one also compliment not only the product but work well with clearly displaying text, making it easier on the eyes and easy to read.

When displaying the product, like on example 2 the products are saved inside cards that have all been placed next to each other with clear information below it showing the price as well as further options for people willing to spend a little more money on their purchase. When it comes to selecting a product, it is not like the previous 2 examples either. The previous 2 examples you would press on the product, which would take you to another page where the images of the watch are displayed on one side and the information on the other side. That is not the case with this application as it only really sells one product.

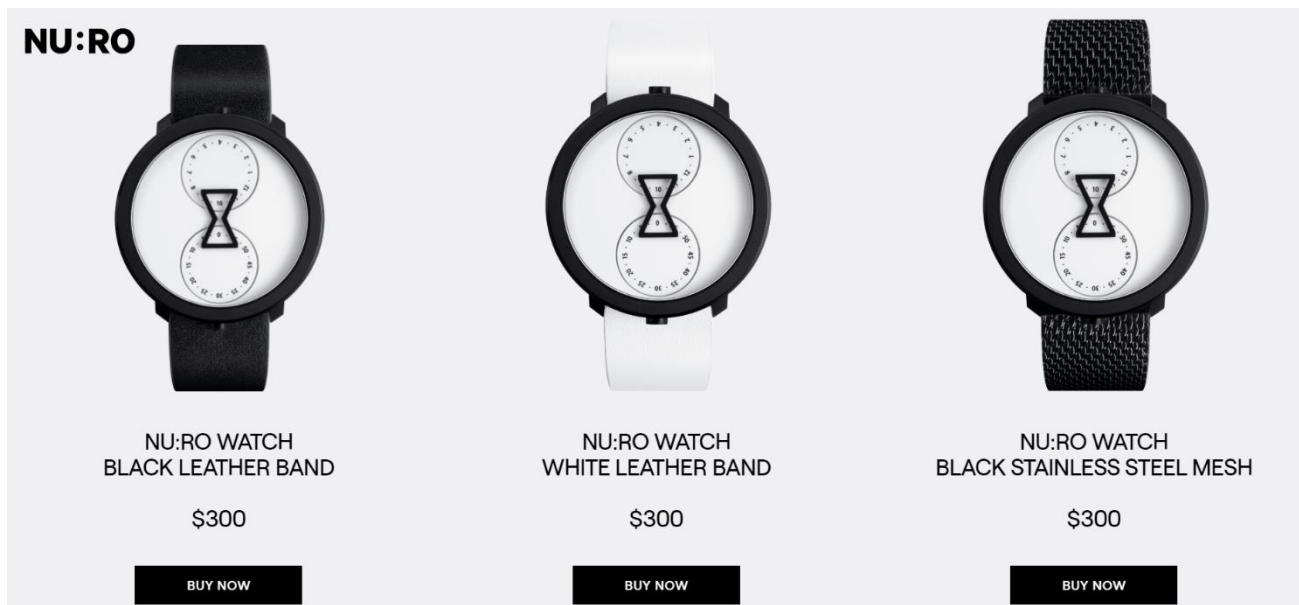


fig 19

So for this application they have added all the necessary information and images of the watch further down this single page to minimise how many different pages a user would have to go through to find the information they need including instruction for the product on how it works, how to read the time of this alternative style of watch. You can see this in fig 20 and 21 below

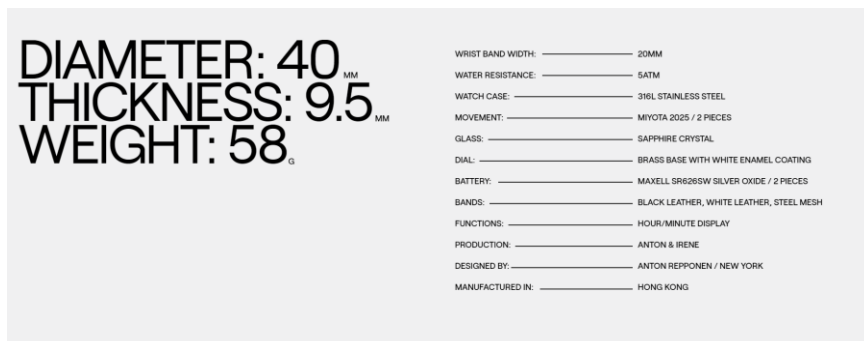


fig 20



fig 21

Nuro.co Functionality – 3.3.2

This SPA compared to the previous 2 has a lot less functionality to it. If anything, NU:RO has the bare bones of functionality with now have more than 1 page as part of there app, excluding the checkout screens. There is some functionality there but it's a lot more basic and the checkout functionality works similarly to example 2 with it's lay out and 3rd party payment options.

How this application works on its functionality is different because of its lack of other pages. Instead links on the screen will automatically scroll you further down the page to the start of the part you are looking for. For example, if you press 'Buy Now' it will take you to the cards with the products on them. It also has an animated home page image at the top showing off the functionality of the watch as well as the different varieties. Further down, there is a watch slide show like that on example one where it shows you the different angles of the headphones. This is towards the bottom but works in the same way. There are buttons on either side of the image allowing you to scroll through the images to get a feel of what the product will look like in hand.

For the checkout the functionality for it is very similar to that example 2. Both in layout, how the checkout process works and it's 3rd party payment options. Only difference really being that example 2 had an option for discount codes, which this one does not, and there is no basket for this application. Once you press to buy one of the products it takes that item and the user straight through to the checkout, do more than 1 product can't be bought at once. Once at the checkout screen the user is able to fill out their details in the clearly labelled input boxes on the left as well as being able to see the product they are buying on the right. The user can fill out all 3 pages of details, that have been appropriately separated into contact, delivery and payment details or the user can press one of the 3rd party purchasing links. They will then be able to log in using there details so this stage is completed for them using the details on the 3rd party app. Once this has been completed the page displays a screen stating what your order is before allowing you to go back to the home screen. The checkout can be seen in fig 22

NU:RO

Information > Shipping > Payment

Express checkout

shop Pay

G Pay

OR

Contact information

Email

Shipping address

First name

Last name

Company (optional)

Address



NU:RO Watch with white leather band

\$300.00

Subtotal

\$300.00

Shipping

Calculated at next step

Total

USD \$300.00

Fig 22

Site used for app research: (NU:RO, 2020)

Tools and approaches - 4

Methodologies – 4.1

During the start of my project I had to think about my approach towards the project as a whole and how I was going to develop it. To do this I began my research by looking up different software development methodologies as well as their advantages, disadvantages and similarities to work out which methodology is best for me. This way I can organise my project a lot easier knowing I what needs to be done first, what needs to be completed by when and what needs to be done to reach the next step. Then by the end I should have a well-constructed program that will have been evaluated and tested thoroughly to make sure I have reached all my aims and deliverables.

After much consideration I have opted to go for the agile methodology. This methodology is aimed around client/user contact to make sure the project outcome I deliver is what the client/user is looking for. In this case the client/user will be looking for a user-friendly e-commerce site that is quick, easy to look at, easy to use and works as they expect from an e-commerce site. By using agile I will be able to create an 'alpha' version of my program which I will then present to a few people, so they can trial the site. After which they will complete an anonymous questionnaire so I can find the good and bad parts of the software. Once this period is complete, I will use the agile methodology again to make amendments before completing and releasing the final version of my project.

Agile consists of several steps as you can see below.

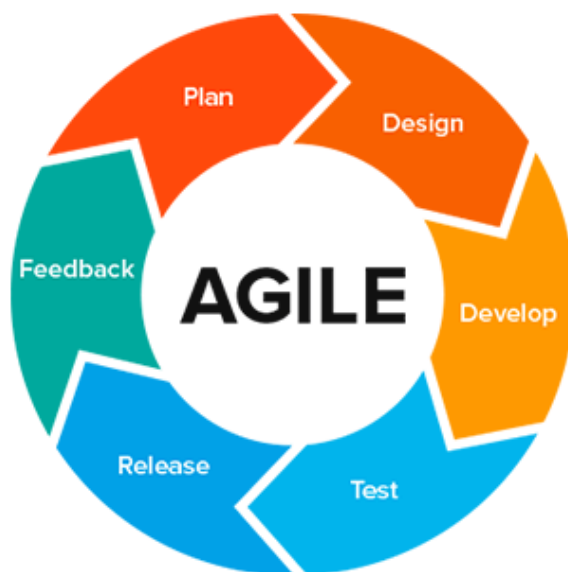


Fig 23

Agile consist of 6 primary steps.

1. Plan
2. Design
3. Develop
4. Test
5. Release
6. Feedback

The Agile process I will be using throughout my project will be the Kanban methodology. The Kanban methodology allows for a steady and organised development of the project at hand by limiting the amount of work is completed at once.

The Kanban methodology works using 3 basic principles:

- 1) **Visualise what to do what day.** – This will allow me to look at all the work in the big picture so I can work out what needs to be done first, when certain things should be completed and how much of a certain task should be completed by the end of the day. It also allows me to keep track of my project a lot easier with it all being in small steps instead of large vague objectives.

- 2) **Limit the amount of work in progress** – This helps balance the flow-based approach so that too little or too much work isn't done at once. It allows for a steady flow of small successes of small targets. Instead of doing sprints like in scrum this method allows me to take my time with the tasks that I need to complete so nothing is rushed. I will never be working on one thing for too long. This helps keep me focused on one thing at a time so things so I don't try do too much at lose track of my target or too little and lose track on how much of the current task I completed and what is left.
- 3) **Enhance flow** – When one task is completed the next one on the list of priorities moves up from the back log to be the new task to be completed. This way I'm only working on what is most needed all the time instead of working on something that may not be needed as much until later in the project.

This encourages active and ongoing learning. Because I will be working on something related to the project each day it allows for continuous learning, learning something new I can apply to the project every day until completion. This will be key when I start to teach myself React.js and Python. Kanban will also show the best possible improvements by working out the best approach to the task at hand.

Information gathered from (researchgate, Mario Špundak, 2014, Mixed Agile Traditional Project Management Methodology – reality or illusion),

(Inflectra, 2020, Kanban),

(Amazonaws, 2020, Comparison between agile and traditional methodology),

(blueprint, 2020, Agile Development 101)

Repository/version Control – 4.2

During this project the user of a repository will be needed to keep track of what part of the program was finished last, to be able to restore to a previous point if the program stops working an a solution cannot be found but also to back up completed sections of the project such as design, back end, implementation etc. For this the GitHub webpage and GitHub desktop application will be used. How this repository will work is there is always a master version of the application. From here the user is able to create branches to work on a different part of the application, without potentially corrupting or breaking the master version of the application. Once the user has completed their branch they can 'commit' the application to the master so that becomes the new master with all the new working functionality added to it. It helps to keep track of versions of the application throughout its development. Hence why the software is called version control.

There are other types of repository like Bitbucket for example. This software does the exact same thing as what GitHub does and could have been quite easily used in place of GitHub. The primary reason for not doing so is for GitHub's software compatibility. GitHub is supported on both PyCharm and Visual Studio Code as a built-in feature. All the user must do is log in and select the right repository they are working on. So instead of having to use a 3rd party application to check for changes, push and pull code to and from the repository as well as manage conflicted merges, the user can do all this using the development software instead. Keeping all the code and debugging nice and neat all in one place.

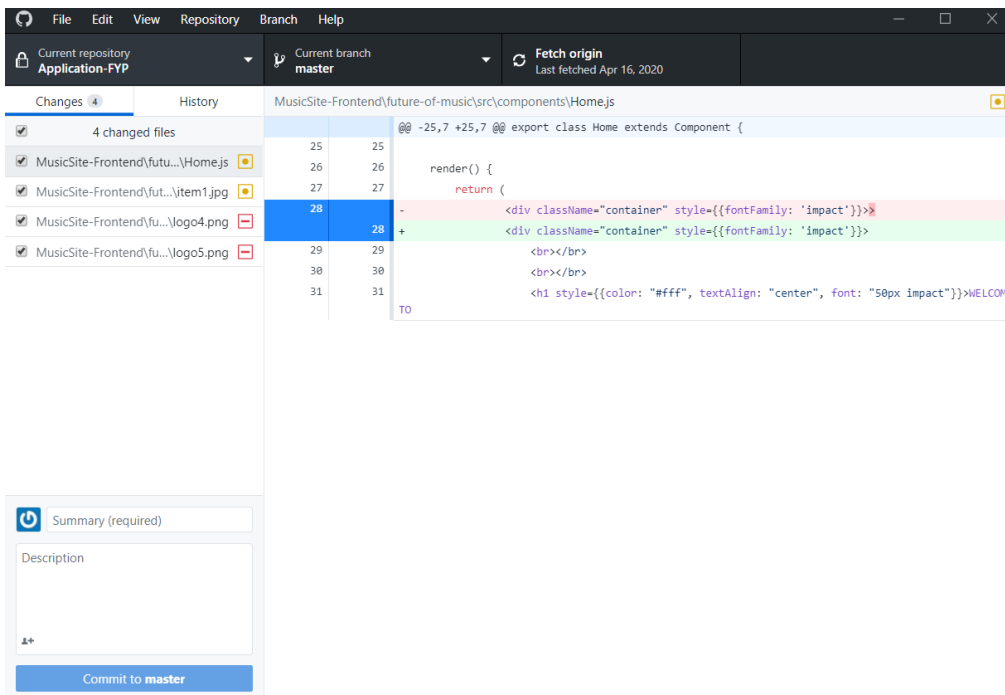


fig 24

As a bonus Repositories like Bitbucket are online Version control applications where the development software has to be connected to it then you will have to access your browser to complete all the functions named above where as GitHub has a Desktop application so if I didn't connect the Version control to the development software I would still be able to do this locally and just push any changes when a stable internet connection has been established. In certain cases, the whole repository can be stored locally so there is no need for an internet connection.

Front end Framework/Libraries – 4.3

When it comes to Front end frameworks to create a single page application a JavaScript framework will be needed that includes a range of smaller libraries that can be installed for it to complete specific tasks throughout the applications functionality. For that reason, the JavaScript framework, React.js, will be used to create the front end of the application.

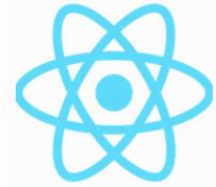
After finding out the single page applications work by changing what is displayed on the screen by reading what is in the URL it was discovered that single page applications are heavily JavaScript and so the framework that would be used would have to be JavaScript. This would allow for live data to be sent from and sent to the front end without the need of web refreshes, a little how Ajax works on your standard multi-page application. There were a few to choose from but eventually it came down to React.js, Vue.js and Angular.js as all of these were JavaScript frameworks and would in turn work perfectly for the single page application.

All 3 of these frameworks or libraries would have worked really well at creating a front end application as they are all HTML based so styling would be easy so a comparison on their functionality had to be made instead about which one would be more appropriate for this type and scale of application.

First, we have Angular. Angular is the more advanced of the 3 frameworks as it requires you to already have a deep understanding and knowledge of JavaScript before hand and so will have quite a steep learning curve if you don't already know a lot about JavaScript and it's frameworks which has been known to turn other developers away from it. Angular is mainly a framework for large scale development teams who already know a lot of Script style languages as it is a very complex but much more complete, due to having most packages built in, than the other 2 frameworks.



Second, we have React.js. React is a much more commonly know framework and a much better framework to be using when first getting started using single page app frameworks and JavaScript. React unlike Angular though is not a complete package. In fact, it is quite the opposite as uses JavaScript on a much simpler level but has the ability to work along side other libraries and packages to create a more complete version of itself. This makes this framework a lot more flexible for developers to use how they want and build it, so it works how they want instead of it already having a complete set of rules like Angular. It also has the ability to seamlessly work along side other frameworks such as Django.py or Express.js which gives even more advantages for developers who want a more flexible application as well as making it future proof for when new frameworks are developed. It gives the developer more range of what they can achieve from it.



Finally, we have Vue. Vue is a much more modern framework compared to the previous 2 and still have a lot of flexibility that can work like React.js. This one does have 1 major downfall though and that is, because of how modern it is it hasn't been used or tested as much as the previous 2 has and doesn't possess as much functionality or accessibility to some frameworks or libraries like React or Angular does. Also with it being so new and not a lot of developers use it, it would make it a lot harder when something goes wrong to find out why as the community behind it isn't as massive as React or Angular so fixing complex problems or trying to make it work along another framework may prove difficult.



All of which have their own positives and negatives but, in the end, React.js seemed to be the best framework for this SPA due to it's massive community which would help learn how to program with it, debug using and create a good front end with a back end with a different framework. Overall, it's the information that is available for it as well as it's huge flexibility and extensibility that React.js will be used to develop the front end.

Information gathered from (codeinwp, Shaumik Daityari, 2020, Angular vs React vs Vue: Which framework to choose 2020),

(Pro Single Page Applications Development, Gil Fink, Ido Flatow, SELA Group, 2014, Using Backbone.js and ASP.net)

Front end Hosting Service – 4.4

For the hosting side, part of what will be used is based upon the language being used in the previous section. Other than finding an external hosting service to host the front end of the application on which will cost money at a lot of extra work when updating the master application the best course of action would be to find a local server hosting application that supports JavaScript. For this reason, I have opted to use Node.js. When React.js was chosen to be used to develop the front end, a server to host this would

have had to have been thought of also as it isn't like a normal HTML page when you can just go to its final directory in the URL. For it to work properly with live data it would need to be on a server of some sort that would allow for data to be sent to it and from it.

With React.js being a JavaScript Framework the server would need to be able to support a heavily JavaScript application and would be able to run effectively with Ajax calls when new live data is being sent through. This way we can minimise the amount of times the web page would have to be refreshed. For this reason, Node.js will be chosen. This is a JavaScript hosting application that allows for users to load web files from it and host them on a local server on your home device. In the long run if this application became live as was going to be used by a business, purchasing a web address and paying for the application to be hosted there instead as this would mean more than people on a local device could see the application. But for the general development Node.js will be used. This way while it's being developed the developer will be able to see the console and its logs in case there is an error. Not only this but when developing a site, you don't want to develop on the primary server, you'll need a development environment first to make sure the application works before being released.

Another reason that will be talked about later is that it is built into Visual Studio Code, the application being sued to develop the front end, so it wouldn't need to be kept open in another window or tab, with Node.js the application can be run or debug from inside the development environment too.

Back end – 4.5

To create a back end application a server-based framework will again have to be used so that the 2 servers will be able to connect to each other over a live connection. The back end will be a RESTful API, so it does not need to be hosted in the same application as the front end. It can run on a completely different server-based application using a different language as the data being sent and received won't be unique to any language, it will all be JSON. So, for the back end the Python framework, Django.py, will be used to create a fully functional API that is able to use the HTTP verbs GET, POST, PUT and DELETE to send or receive information via requests sent from the front end. To be able to run the Django back end server the application will have to be run from a console, just like Node.js so, to avoid confusion this application will be run in a virtual environment using PowerShell. Also, Node.js is for Java based application and this application is entirely Python.

Another application to complete this back end RESTful API would be express because of its Node.js support, its extensively libraries that can be used with a simple install as well as its compatibility with IO

to create things such as a live chat system and the sheer level of flexibility of the application. Plus having an application in the same language as the client side, React.js, makes for better scalability and easy development with a team of JavaScript developers. On the other hand, for someone who doesn't know a vast amount of JavaScript and doesn't have a team behind them then using express can be very complicated and get very messy with all the files you'd need to create connections, promises and having to create classes that work when certain commands are received. This is not the case for Django.

Django has a set file system for creating things like connections and classes where the developer will have to go through a series of files to add and edit code to create this system. This sounds complex and it is if you have little understand of python, but once the developer knows what to do the scalability, the ease of development increases massively. So once a developer has full understanding of Django then they will be able to develop API connections, database connections as well as requests quite easily allowing for the rapid growth of the back end system. Not only this but Django has some features built in that Express



doesn't. For example, for a select number of security measures to be involved with Express.js the external package 'passport.js' would need be installed, integrated and understood to have a basic level of security to it. This is not an issue with Django as it contains some onboard security features. Such as for password. In the Model for the database if the entry point is specified as a password or another feature that may need security measures round it, Django will automatically encrypt the passwords instead of needing an external package.

For its scalability and inbuilt security features as well as it's easy of development once a deep understanding of the framework has ben learnt I will be using the Django framework to create my back end API. The file format and layout make it clean code to be using too as all components of the framework are clearly displayed and named creating a tidy in house application instead of having to spend time making the overall structure of the back end look tidy



Information gathered from (For Seeromega, James Burns, 2019, Django Vs Node.js: A Detailed Comparison, Pros and Cons),

(data-flair, DataFlairTeam, 2019, Django Advantages and disadvantages – Why you should choose Django)

Database & Database Tools – 4.6

For storing data for this application, the application will store it's data using the MongoDB NoSQL database. NoSQL is for no relational databases where often large amounts of data are stored. This is a positive as a full-scale e-commerce site would have to store enormous amounts of data at a time such as products, profiles, purchases etc. All these tables would all be required to be updated in close proximity to each other. Another reason for using the NoSQL database MongoDB is it's use for real time apps. A real time app is where data on the screen of the application changes live when the data in the database does. A little like how it does in single page application when it updates the user's basket, or the quantity of the item being bought. This also uses a wide range of different database technologies instead of using the normal SQL syntax, NoSQL can store structured information instead such as arrays of data or polymorphic data. All this along with MongoDBs scalability and it's high performance and flexibility is why MongoDB will be used for this application

Along side MongoDB I will be using the application MongoDB Compass. This is a partnering tool that works when you connect it to the MongoDB server on Nodes.js will be able to display all collections, tables, rows of data and collections of data all in one place. It also allows the developer to edit and delete the data in MongoDB as MongoDB on its own does not have a user interface and would have to locate all the data in the DB (database)

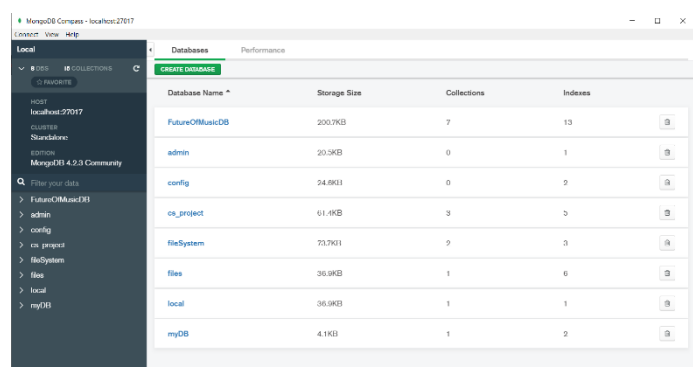


Fig 25

through console commands instead. Using the compass, the developer will be able to locate data edit it and delete it very quickly and very easily.

Information gathered from (Guru99, 2020, SQL vs NoSQL),

(Clariontect, Rakteem Barooah, Applications that work best with NoSQL)

API Endpoint Testing Software – 4.7

To test endpoints of the application the developer could just enter the request into the browser to see if the data is being received by the server. But testing the end points require more than just receiving data. So to make sure the API can successfully send, and receive data into all databases, the best course of action is to use a API development collaboration platform. For this Postman will be used. Postman will allow the developer to enter all the different types of requests into application to check if the data is being sent when POST, PUT or DELETE functions are used as well as being able to get data when GET for all items or for specific circumstances like when a search is completed. Not only this but if there is an error it will clearly display in the body what it is and a vague idea on how it can be fixed. This will also store past requests so you don't have to keep typing out the same requests over and over, you can load an old request and try it again. Finally, it can also be used populate the database when the developer hasn't created a way to input certain data on the front end. This way the data reaches the database in the same format as the rest of the data. All data from the request is received in which ever format the developer wants or that the API can send. More than often it is in JSON but it can also receive data in HTML and XML.

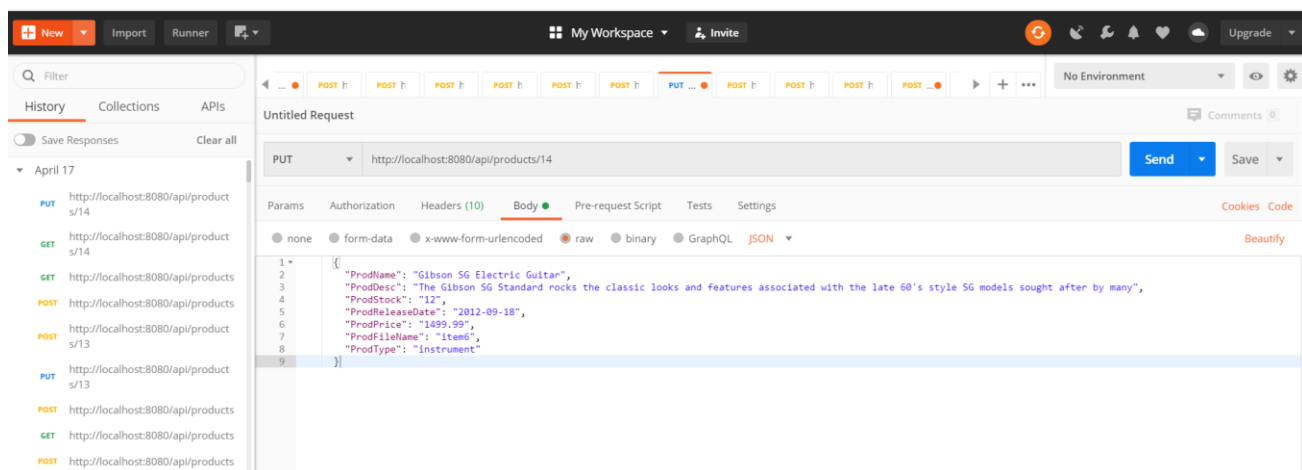


fig 26

Application Development - 5

Requirements– 5.1

Before the development of the 'Future of Sound Online Retailer' could begin, the basic functionality for the application or requirements had to be identified, how it would work, and what will the user need to be able to do at the absolute bare minimum to be able to complete a transaction on the site or to be to complete a range of tasks capable of being done on Amazon also so at the end the user testing can compare the 2 applications effectively.

These Requirements are:

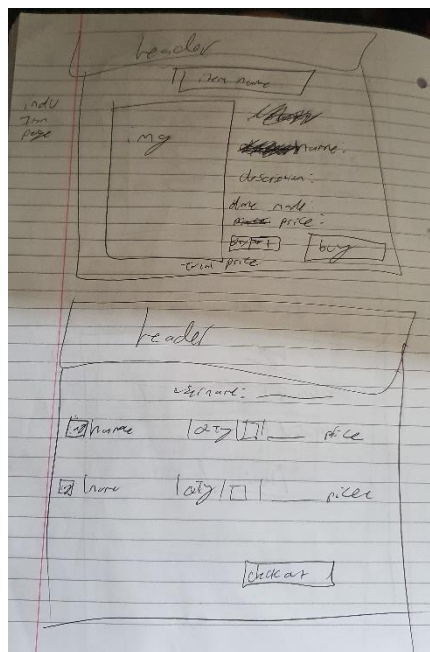
- The application must be built using only 1 HTML page
- The application must display products in a list format for users to be able to browse the products being sold by the website
- The application must be able to retrieve information about the products from the Mongoose Database via the back end API system
- The application must be able to connect to the API view API requests (GET, PUT, POST, DELETE etc)
- The application must be able to connect the database to the front end via the back end API
- The application must be able to display a more detailed product when a listed product is pressed. The page for the item cannot be hard coded and must be dependent on the items ID so that the same page is loaded with different data every time
- The application must allow the user to add items selected to their basket at any quantity available up to 5
- The application must allow the users to view all items added to the basket at one time
- The application must allow the user to delete items from their basket
- The application must allow the user to checkout using data available to them. (**Note** personal information is needed to complete a transaction but for user testing, users will not be using their personal information. Information will be provided for them)
- The application must clearly display when a transaction or other task has been completed.
- Information from this application must be stored in the correct tables on the database
- Information from the database must be send to the front end in JSON format
- The API must format its data for the database. Incorrect data will not be sent to the database
- All 3 components to the application must be hosted on a server-based application (Node.js, Django.py)
- The Front end application must be aesthetically pleasing to the user, easy to use, easy to navigate and must not cause the user any discomfort when being designed

Pre-Development Design– 5.1

Before the application development could begin a general idea or design would need to be created so that when building the front end, I knew what I was aiming for. This were not detailed drawings as these were only quick sketches to get a general idea of what the application was going to look like to the user. It

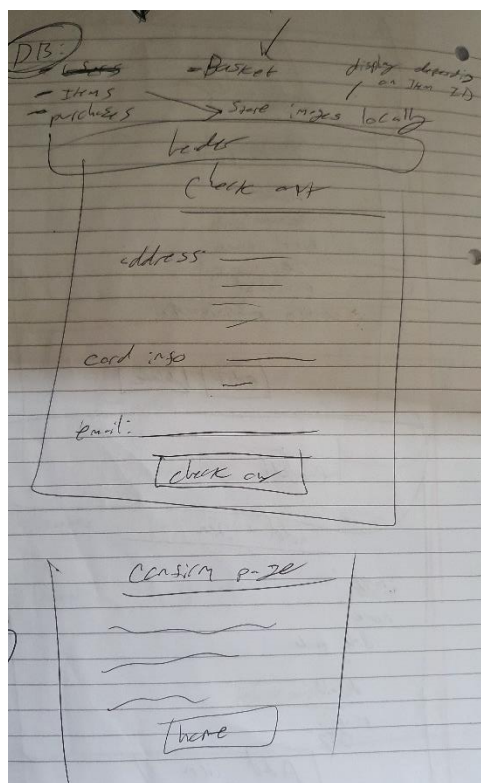
contains each of the major pages available with a few quick sketches inside of each one to represent a part of each of the pages.

The idea behind these drawing was all linked the combining the 3 examples listed earlier in the report (4.1, 4.2 & 4.3) as all 3 of these examples had very modern but very similar UI design that worked throughout all of these pages minus the odd component having an exception on each of the pages. For example when designing these pages I took into account the header of both pages 1 and 2. These headers at the top of the page showed off the general pages available to the user as well as what was available while staying professional with a well contrast colour scheme and a professional looking logo. So because of this in my initial designs I made it clear how I was going to add a header which would be displayed at the top of each of the individual pages.

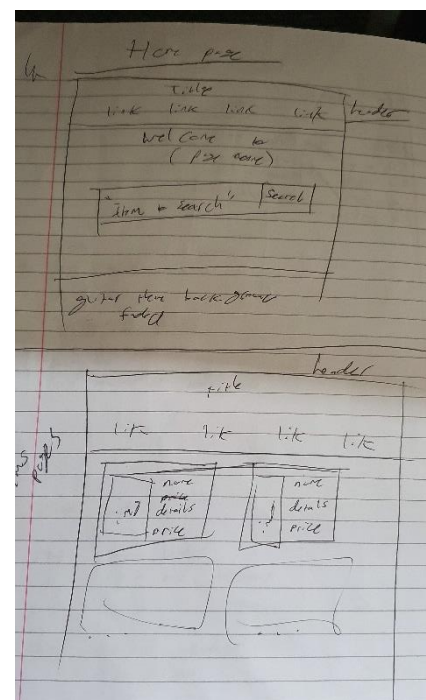


Not only this but all 3 examples had a background image on the very first page to show off the product(s) they were selling as at least a picture that would represent the site itself. So again, I have opted to use this with this e-commerce site as I have made a note in my designs that a music-based background them in them. On example 2 and 3 this was just the price and a button to either buy the item or to do to another page to see a more detailed version of the product. I decided to use these cards for exactly that reason as it gives a clear view for the user of the product and what it is when they are scrolling through a lot of different products. If all the information was on each of these items, it would take much longer to find the product and may turn users away.

The final feature I took from the examples for my general design was the item page itself. The item page displays the product on one side of the screen and all the information,



including price and quantity on the right. All information clearly labelled showing off the product as best as possible. This primarily was a combination of example 1 and 2 as these both had pages such as this where on the right was an image of the product available and on the left was all information based around the product the user would need such as title, description, price and an option to choose the quantity of the product the user would want to purchase. This made searching products much easier and clearer for the user instead of just using an image and a price such as example 3.



Front end Development– 5.3

The development of the front end started out as a Node.js application that first had to have a few different packages installed into it to allow the application to have the full functionality needed by it to complete the requirements listed above. These packages were React-Server-Dom so that links between pages were a possible thing as normal React doesn't have this function. The React framework to develop the front end with, bootstrap to help with the design of the pages through the use of its onboard classes and finally awesome fonts. This was a simple package to primarily just introduce user icons for the pages such as a 'basket' icon for the basket link. This proved to be slightly more tedious than expected as some of the packages that needed to be installed had '-S' on the end and if you used a lower case one instead it would install different files to the ones needed. Also, I couldn't do more than one at once. For this to work they had to be installed in a certain order (React, server-dom, bootstrap, awesome fonts) otherwise the files wouldn't load properly and some of the packages would error on installation because others were missing.

Once this was set up it was onto the development of the actual application. To start off with the application for my own reasons had to be running first. The application updates the server and the front end every time a new change is made and will display errors on the page when there is one. So this was visible and I got a fast response from the front end when designing the application I opted to start the server first with the npm start command.

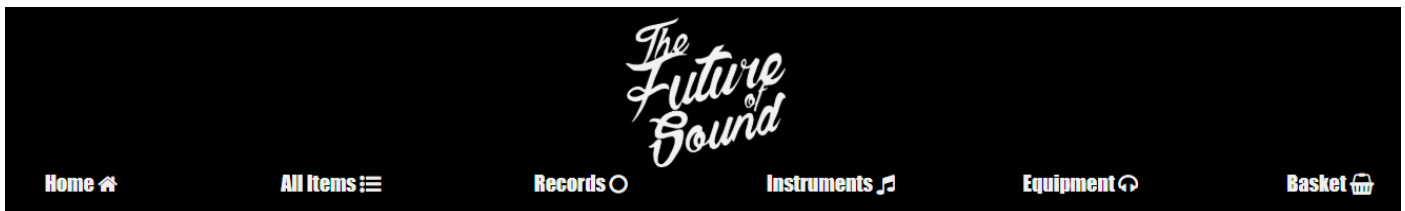
Now that everything was running, I had to set up the application's directory App file. This is where all the different files would be called from when links will be created later on. It's also what is rendered in the single page part of this application. This was just the case of importing all the pages I could be using during the development of the app as well as the header. Each of the pages were also given a specific path and linked to the component that they would be rendering in. the 'path' would be what extension to the URL would be needed to load a specific page such as /home would load the home page when a link to /home was created.

When starting the development of the front end I thought it best to work on the general look of each page, what will be the exact same on every page, and build that first. This way building the rest of the application would require less repetitive code. So to start off I developed the header for the application as this would be displayed on all of the pages at the very top. This was developed using the bootstrap package and the bootstrap classes to create a simple header containing the links to all the pages possible, with the link component from react-dom to change the URL, that would be easy to read and contain the web sites logo as to show of the company at all times. This also included icons for each page brought in from the awesome fonts package for a visual representation of what each page contains for those who may struggle with reading the text in the links. The difficult part to this was the positioning of the logo. Without logos must be included in react so the image appears its positioning became an issue as it would keep overlapping with other components, be 1 pixel thin or appear in the wrong part of the header. This was solved by adding it to its own div and centralising it above all the other parts of the header and fixing the images width so it couldn't get so big it messed with the styling of everything else.

```

return (
  <div style={headerStyle}>
    <div>
      <Link to="/home"><img src={require('./imgs/logo3.png')} style={{width: "8%}}></img></Link>
    </div>
    <div className="container">
      <div className="row">
        <div className="col-md-2"><Link to="/home" style={linkStyle}> Home <i className="fa fa-home"></i> </Link></div>
        <div className="col-md-2"><Link to="/all-items" style={linkStyle}> All Items <i className="fa fa-list-ul"></i> </Link>
        <div className="col-md-2"><Link to="/records" style={linkStyle}> Records <i className="fa fa-circle-o"></i> </Link>
        <div className="col-md-2"><Link to="/instruments" style={linkStyle}> Instruments <i className="fa fa-music"></i>
        <div className="col-md-2"><Link to="/equipment" style={linkStyle}> Equipment <i className="fa fa-headphones"></i>
        <div className="col-md-2"><Link to="/basket" style={linkStyle}> Basket <i className="fa fa-shopping-basket"></i>
      </div>
    </div>
  </div>
)

```



The other part to this would be the background as this would also have to be on every page and must sit behind all of its content. This was introduced by importing the image first and then using its variable in the src part of the image tag used in app.js. This way it would load properly and always be in the background so it wouldn't have to be re-rendered on each page. This did have its issues though as to start off with all the text then create from other pages, including the header, would appear underneath the image instead of overlapping it. At first, I thought this was due to the image being larger than the screen. This was fixed by adding its width to 100% so it couldn't be larger than the container it's in. This didn't fix the problem. As the page information was still being displayed below. Instead to fix this I added positioning properties to both the background and the fore ground of the page. If I fixed the background in place and made the foreground relative to the page it would finally be displayed over the top.

```

<Router>
  <div style={{backgroundImage: `url(${backImg})`, maxWidth: "100%", minWidth: "100%", height: "100%", position: "fixed"}}/>
  <Header />
  <div className="container" style={{position: "relative"}}>
    <Route exact path="/" component={Home} />
    <Route path="/home" component={Home} />

    <Route path="/item/:id" component={Item} />
    <Route path="/all-items" component={AllItem} />
  </div>
</Router>

```

The individual pages again were developed using the bootstrap package. More specifically the delete page, the individual item page and the grouped items pages were developed using bootstraps cards class as this allowed me to sort an image apart from the text needed for each of these cards. Cards can be split up into 2 sections allowing images to appear on one side and text on the other. For the grouped items pages I built it so that the cards are smaller and appear in rows so that a few items can be seen by the user at one time. This also covers a small amount of data for each item and an image. Each of these cards were then stored inside link tags so I could turn them into links when the card is pressed it would then pass the items ID through to the link and direct to the individual item page. The main issue with this was when cards appeared directly under each other. This was a simple fix as all I had to do was add a sensible amount of padding to each of the cards on the bottom.

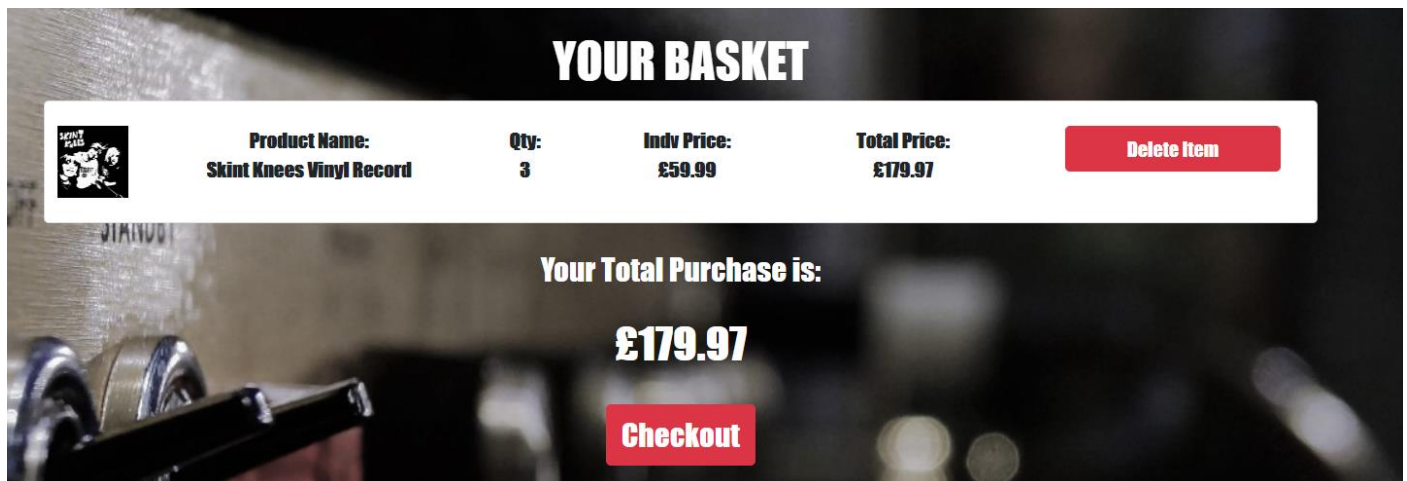
The main problem with building these cards was the lack of knowledge about the latest version of bootstrap. Before in bootstrap these card classes were called panels. So, when it came to try to build one I was using old commands instead of the new ones. Before I realised this, I was using the class name panel, panel-body to test the panels. Next would appear but not in the format required and was unclear why. It was only after a little research into the latest bootstrap update did, I realised that panels had been replaced by card. Once this was known and implemented the cards would build as they should have originally

```
<div className="col-md-6" style={{paddingBottom: "20px", fontFamily: "impact"}}>
  <Link to={"item/" + props.item.id} style={linkStyle}>
    <div className="card" style={itemCard}>
      <div className="row">
        <div className="col-md-6" style={{height: "200px"}}>
          <img src={{images[props.item.ProdFileName]}} style={{height: "100%", width: "100%"}}></img>
        </div>
        <div className="col-md-6">
          Product Name: {props.item.ProdName} <br><br>
          Price: £{props.item.ProdPrice} <br><br>
          In Stock: {props.item.ProdStock}<br><br>
          Click for more product details...
        </div>
      </div>
    </div>
  </Link>
</div>
```

The individual page was developed in a very similar way in the way cards used. Only this time the right-hand part contained a lot more information that would be pulled from the database later on. But so the users could buy these products I had to add into this page a simple drop down that contained the numbers 1 – 5 that allowed the user to select a quantity and a button to add this to the database. This was all contained in a form so that it could be submitted on press. There were 2 small issues encountered with this. One was the shape of the images and the other was button. First, the image. The image if too wide would stretch into the other half of the card and would look very unappealing. To avoid this in the future I fixed the size of the box as well as the image tags so that it cannot expand more than the width of its container. The button on the other hand was less of an issue and more of a design fault. The button originally had on it 'Submit' which seemed passive aggressive as well as unclear or what it would submit to people unfamiliar with it. Unfortunately, I was not sure on how to fix this as doing in the same style of a regular button caused a huge design error on the front end. In the end it was found out that I needed to add the next for the button inside the value field. Using placeholder and adding text in between the input tags either changed nothing on the screen or cause errors.

```
<div className="row">
  <div className="col-md-6" style={{height: "475px"}}>
    <img src={{images[this.state.ProdFileName]}} style={{height: "100%", width: "100%"}}></img>
  </div>
  <div className="col-md-6" style={{height: "475px"}}>
    <strong>Product Name: &nbsp;&nbsp;&nbsp;{this.state.ProdName}</strong><br><br>
    <strong>In Stock: &nbsp;&nbsp;&nbsp;{this.state.ProdStock}</strong><br><br>
    <strong>Description: &nbsp;&nbsp;&nbsp;{this.state.ProdDesc}</strong><br><br>
    <strong>Date Released: &nbsp;&nbsp;&nbsp;{this.state.ProdReleaseDate}</strong><br><br>
    <strong>Price: &nbsp;&nbsp;&nbsp;£{this.state.ProdPrice}</strong><br><br>
    <select onChange={this.onChangeQuantity} required="true" className="form-control" style={{width: "160px"}}>
      <option value="-1" selected disabled>Quantity</option>
      <option value="1">1</option>
      <option value="2">2</option>
      <option value="3">3</option>
      <option value="4">4</option>
      <option value="5">5</option>
    </select>
    <label><strong>Total: <br></strong><input className="form-control" disabled value={"£" + parseFloat(this
    <input type="Submit" className="btn btn-danger" style={{fontSize: "20px"}} value={"Add To Basket"}></input>
    <br><br>
    <img src={require('../imgs/logo.png')} style={{width: "20%", height: "20%", bottom: 0, right: 0, position: "a
  </div>
</div>
```

When it came to the basket on the other hand this was developed in a very similar way to the cards on the grouped items page. As these are again just cards containing images and information that will be brought through using the API. The only difference being that I have designed these ones to take up a whole row so the user can see more of the information they might need about the product and to add a button on the end that will later allow the user to delete a product from the basket if needed. This is designed in a list style so they are able to through all their products with a grand total at the bottom so they can see their total order before pressing the checkout button to pay for their goods. The only real probably with this was the col-md class for the cards. It needed a little more work towards to make sure that they could all fit the right amount of data in without it being too cramped. These ended up being a range of col-md-1, 2 and 3s as these were small enough to fit multiple on as the max total is 12 as well as still giving enough space for some of the larger fields.



Finally, the checkout page. Development for this page was a lot similar than that of previous page as this page was made to be as simple as the possible for the user as possible. Checkout was developed by using the bootstrap class 'form-control' on a range of input fields that would be used as part of a form to upload data to the database. Each of the input fields would contain a placeholder so the user knew what it contained as well as a text limit in some of the boxes so that string messages weren't too long to be sent to the database via the API. This was quite straight forward and allowed for an easy completion of a transaction later on in testing.

Although a problem was discovered later with some of the input fields. Most of the number input fields allowed for some string characters such as e or + in them and these could not be allowed for validation purposes as well as data purposes due to the fact if this application was real it could prevent a user from completing a transaction properly or even stop them from receiving their products. To avoid this, I implemented another package called 'react-number-format'. This allowed me to create a different type of number input field that included a range of validation to it such as what characters had to be included, how many digits were allowed and in what format the numbers would be saved. This allowed me to format the card details into 4 sections of 4 digits only, the CVV number to just 3 digits and the phone number to include the + at the start and no other string value so that the user had to implement their countries code

as well, if the order would be international.

```
<label style={labelStyle}>Surname:</label>
<input value={this.state.PrchSur} placeholder="Surname" className="form-control" style={dataInputStyle} onChange={this.onChangeSur}></input>

<label style={labelStyle}>Email:</label>
<input value={this.state.PrchEmail} placeholder="Email" className="form-control" style={dataInputStyle} onChange={this.onChangeEmail}></input>

<label style={labelStyle}>Mobile Number:</label>
<NumberFormat value={this.state.PrchMobile} displayType='input' thousandSeparator={true} placeholder="e.g. +44" className="form-control"
style={dataInputStyle} format="#####" onChange={this.onChangeMobile}></NumberFormat>
<br></br>

<label style={labelStyle}>Address Line 1:</label>
<input value={this.state.PrchAddr1} placeholder="Address Line 1" className="form-control" style={dataInputStyle} onChange={this.onChangeAddr1}>

<label style={labelStyle}>Address Line 2:</label>
```

Email

Mobile Number:

+447876677876

Address Line 1:

Address Line 2:

Address Line 3:

Town/City:

Region/State:

Further details on exactly how the front end was developed can be found in Appendix 10.2.1

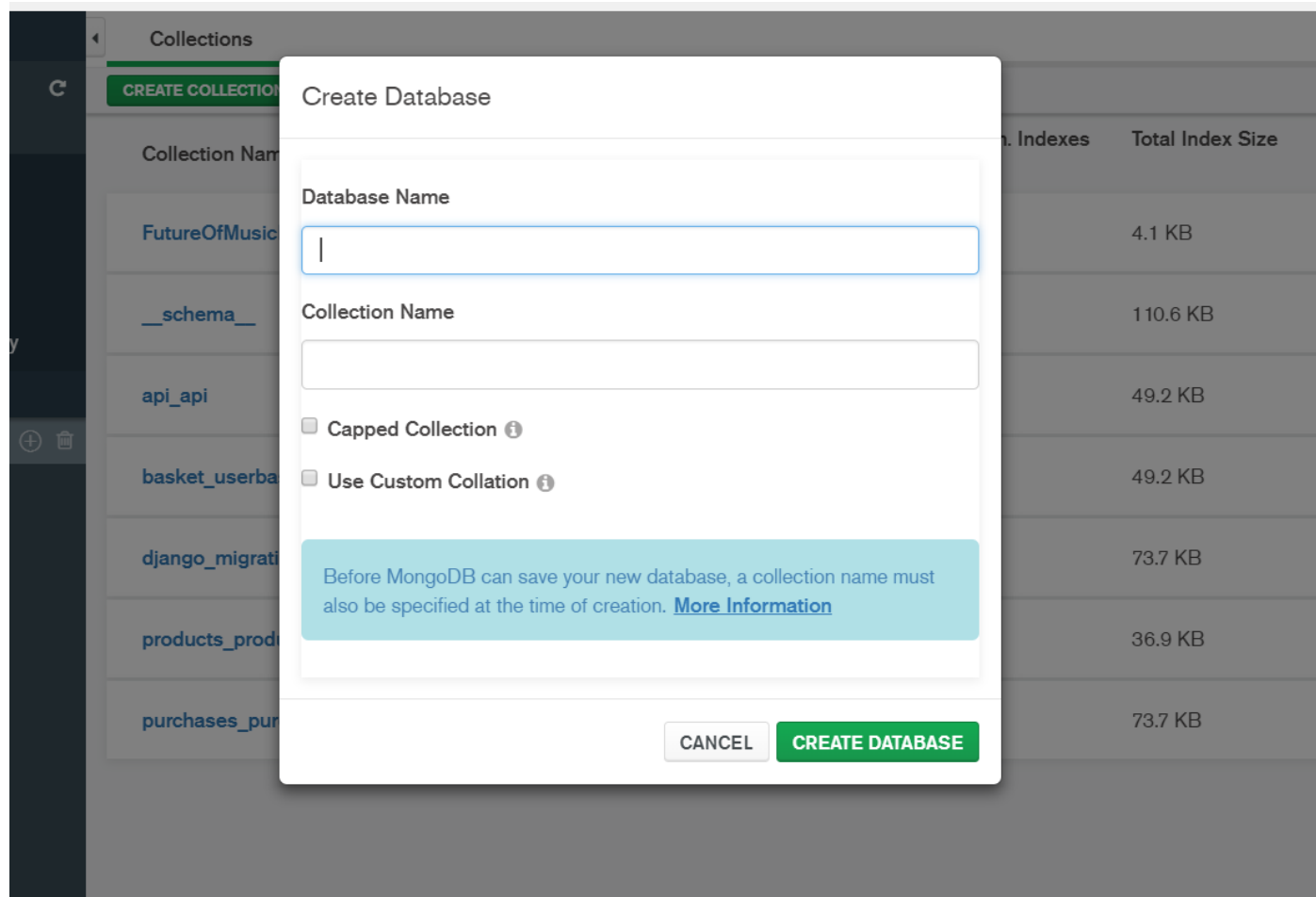
Database Setup— 5.4

The development of the database was quite easy and straight forward as there was very little development to do with setting up the database. The database only really consisted of downloading the files from the MongoDB website, renaming the database file, adding in a data folder with another folder named db inside of it. Once this was done all that was needed was run the exe file from a node server so that the API could access it.

```
2020-05-05T18:05:59.511+0100 I STORAGE [consoleTerminate] Finished shutting down checkpoint thread
2020-05-05T18:05:59.526+0100 I STORAGE [consoleTerminate] shutdown: removing fs lock...
2020-05-05T18:05:59.529+0100 I CONTROL [consoleTerminate] now exiting
2020-05-05T18:05:59.530+0100 I CONTROL [consoleTerminate] shutting down with code:12
PS D:\FYP\Application> C:\Users\steve\Desktop\MongoServer\bin\mongod.exe --dbpath=C:\Users\steve\Desktop\MongoServer\data\db
```

Once the database was hosted then I could use the MongoDB Compass application to create a database that would be used to store all the different tables that will be used by the application. Because this app has a simple UI it was clear to see how to connect to the server for MongoDB as well as clear to see how to simply create a database. The only real issue with this part would be the creating the database. It was unclear on what needed to be added in the collection name input field. This was solved by entering the same as what was in the database name field. After this the database was set up ready to use.

Alternatively I could have completed these tasks using commands in the command prompt for the database but this would have required a lot more work to complete a small task and a lot more things can go wrong as minor things such as spelling errors or grammar errors in the capitalising could cause a lot of errors down the road. Plus, editing data could be a lot more difficult than just clicking edit on an item and adding the new data in.



More detailed instructions about how to set this up can be located in appendix 10.2.2

Back end Development– 5.5

Developing the backend was a lot more complicated and file hungry compared to the previous development areas due to it being quite a large collection of files and folders for each database as well as how much work goes into the preparation before the development can start. For starters I had to create a virtual environment to create the application so that anything installed for the application doesn't potentially damage any systems files due to it being built from the PowerCell which has access to the system 32 files.

Once this was complete the next step was very similar to that of the front of the application as again this application is very server based and requires quite a range of different packages to be able to run successfully. First of all, once python has been installed, I had to install pip as this is the python package manager that will help manage any packages for the app. After this I needed to install the packages Django, djangoestframework, djongo (not to be mistaken with Django) and the CORS package.

```

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'api.apps.ApiConfig',
    'products.apps.ProductsConfig',
    'basket.apps.BasketConfig',
    'purchases.apps.PurchasesConfig',
    'consheaders',
]

```

Upon installation of the package I was able to start building the API application from the console by using startproject MusicSite to create folder where the app will be created. This like react built the basic file layout and functionality read to be developed. Now I had access to all the files I was able to include all the files needed in the installed apps section of the app so the app knows what packages it can use. There was a different include for each of the packages other than the normal 'Django' package as this is what was used to build the files in the first place. This caused a few errors early in the development as errors were showing up for reasons unknown because I forgot to add some of them to the installed_apps so until I did this I was limited with functionality.

Now that all the right packages were included and ready to use I had to start out by connecting the MongoDB to the Django API application using the djongo package. This was quite straight forward as there was already a section created in the app for the database and all I needed to do what edit this to the MongoDB server I have developed earlier on using the same port, host, database name and include the engine (Djongo). This would then allow the application to connect to the database.

```

DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'FutureOfMusicDB',
        'HOST': '127.0.0.1',
        'PORT': 27017,
    }
}

```


The next step was to add a layer of security to the application so that only certain Ports were able to connect to the API as well as make sure that the front end of the application can make requests for data. For this I used the CORS package. This brought up a few issues I will mention later. If CORS was not installed onto this application it means anyone who would have access to my network would be able to access the API, make requests and retrieve potentially very personal information such as card details, addresses and passwords.

Once the CORS package was added to the installed apps it also needed a few other settings changing so that it would work with the front end. This included adding cors.headers to the middleware so that it would check the CORS security before everything else as well as adding the front end port to the white list so that it is able to make the necessary requests. A problem was brought up here as my original thought was that the port needed on the white list I first believed to be for the API so that it was visible to other apps where in fact it needed to be the front end port so the API knew which external app it was allowed to connect to. This was fixed when errors appeared on the front end when the request was made, and the console displayed the CORS header origin to be the problem. Allow all is false so that only the whitelisted apps can access it

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'corsheaders.middleware.CorsMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]  
  
CORS_ORIGIN_ALLOW_ALL = False  
CORS_ORIGIN_WHITELIST = (  
    'http://localhost:3000',  
)
```

The set up for the API application was long and frustrating with how many things needed to be installed and set up before it could all start but in the end it allowed for a safe and secure application for the 'Future of Sound' application.

Once this was complete the basic functionality of the API was ready to be created through the use of project apps. These were smaller folders inside the initial project folder that will contain different sub applications that will connect with different databases and handle different information like controllers. So to start I had to create this sub apps using the startapp products command. This would then create a sub application called products. This was done for the basket and purchases apps as well as these are the 3

different database that were used in the overall application. Once these were made, I had to include there config files into the installed apps like the packages so the application knew to use them.

Once this was done for all the apps, I was able to start developing the models, serializes and views so that the API knew how to handle certain data from different databases. This is what is used to transcribe the data from the database tables into json for the front end to handle. It also creates the functionality for what data is brought back when specific requests are made.

These all started out in the same way. First thing that had to be done was to create the model for each of them so the app knew what data it was going to be using including validation to it so that the application knew what it should allow. Examples include character limit, if it could be blank or not, it's default value and how my decimal places a number had to be rounded to. There was a problem using this with dates when working on products as the datetime function would not work when entering the data for its default value and prevented me from moving forward from this. I could have opted to use just a normal string to do this instead but to make sure that the data format and style is kept the same so that the database only accepts the right data I opted to use the DateField instead with a datetime stamp function.

This was due to my input of the date. I was using a 2-character date for the month and day that was not needed. Using '09' for September was causing the issue when it only needed to be '9'. Once this was done the model was migrated to the database to automatically set up the tables with the validation in place ready to be used.

```
class Product(models.Model):
    ProdName = models.CharField(max_length=100, blank=False, default='')
    ProdDesc = models.CharField(max_length=250, blank=False, default='')
    ProdStock = models.DecimalField(max_digits=2, decimal_places=0)
    ProdReleaseDate = models.DateField(default=datetime.datetime(1997, 9, 16))
    ProdPrice = models.DecimalField(max_digits=7, decimal_places=2)
    ProdFileName = models.CharField(max_length=70, blank=False, default='')
    ProdType = models.CharField(max_length=20, blank=False, default='')

```

Next, I had to create serializers. This is what could turn the database information to JSON to be handled on the front end and convert the JSON sent from the front end to a set of values to be entered into the database. This file was easy to build as all it had to include was what model the serializer should be working from so it knows what the validation is for all the input fields as well as a list of field names for the database. These had to be in the same order as those in the model or the serialize wouldn't work. This way when the data reaches the serializer it knows how it should appear in the database and is easily broken down into JSON when data is being retrieved.

I opted to use JSON is it is a lot more universal than that of XML and is a lot easier to handle in the front end using react. Although XML could have been used it is a lot more difficult trying to break it down as it uses HTML elements.

```

from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = (
            'id',
            'ProdName',
            'ProdDesc',
            'ProdStock',
            'ProdReleaseDate',
            'ProdPrice',
            'ProdFileName',
            'ProdType')

```

The final stage of creating this API was the functionality for when certain requests are requested. First of all I had to define the request urls in the urls file so that app knew what function to complete when a certain request is asked for by the front end.

NOTE: the app also needed to be added to the projects urls so that it knows to use these request and functions when asked for them. Project urls on left, app urls on right

<pre> from django.conf.urls import url, include urlpatterns = [url(r'^\$', include('api.urls')), url(r'^', include('products.urls')), url(r'^', include('basket.urls')), url(r'^', include('purchases.urls')),] </pre>	<pre> from django.conf.urls import url from products import views urlpatterns = [url(r'^api/products\$', views.product_list), url(r'^api/products/(?P<pk>[0-9]+)\$', views.product_detail), url(r'^api/products/instruments\$', views.product_list_instruments), url(r'^api/products/equipment\$', views.product_list_equipment), url(r'^api/products/records\$', views.product_list_records)] </pre>
---	---

This way request knows what to include when requests are sent such as if and ID is sent or a category for a product in this case. Once these were set up the views or functions could be built in the views file so that the app knows what to do or how to handle data when one of them are requested.

The views, like the rest of the API were developed using python and each of the functions, as listed in the app urls above, were create inside views using a set of nested if statements that will return certain values depending on the values passed or not pass to them. For example. View.product list when used with GET

will collect all the products and info for them from the database where as a POST would add data to the database instead. Each of these functions use the serializer and the model so it knows what model it will need to use and then serialize the data before sending the information through a return to the front end.

The `product_details` function is an ID based function created where it will use the ID of a product in the database to find all the information about that one product from the database if a GET is used. If PUT is used to will allow the admin to edit the information and using DELETE will delete this one item.

The 2 types of functions listed in the previous tasks are used through out all other models so the API is able to add data to the basket table and purchases database as well as delete it when needed. It also allows admins to edit this information using Postman.

```
@api_view(['GET', 'POST', 'DELETE'])
def product_list(request):
    if request.method == 'GET':
        products = Product.objects.all()

        ProdName = request.GET.get('ProdName', None)
        if ProdName is not None:
            products = products.filter(ProdName__icontains=ProdName)

        products_serializer = ProductSerializer(products, many=True)
        return JsonResponse(products_serializer.data, safe=False)
        # 'safe=False' for objects serialization
    elif request.method == 'POST':
        product_data = JSONParser().parse(request)
        products_serializer = ProductSerializer(data=product_data)
        if products_serializer.is_valid():
            products_serializer.save()
            return JsonResponse(products_serializer.data, status=status.HTTP_201_CREATED)
        return JsonResponse(products_serializer.errors, status=status.HTTP_400_BAD_REQUEST)

# GET list of tutorials, POST a new tutorial, DELETE all tutorials

@api_view(['GET', 'PUT', 'DELETE'])
def product_detail(request, pk):
    # find tutorial by pk (id)
    try:
        product = Product.objects.get(pk=pk)
        if request.method == 'GET':
            product_serializer = ProductSerializer(product)
            return JsonResponse(product_serializer.data)
```

The last 2 functions that are unique to products, add a product type to the end of the request. These are simple GET functions only and they will automatically get all data from products with the product type of Record, Equipment or Instruments.

Another unique part of products is it's such abilities. The first get function, if passed a phrase in, will search for all items with a title like what has been searched for. This was added to the search bar on the home page of the application.

Once all functionality was written for the apps in the project with working serializers this could be tested using the Postman application to make sure the right data is being retrieved and sent before connecting it to the front end. This will be mentioned later on in the testing chapter. For the testing to be done and so the API can be used to API and its server need to be activated. This was done using a simple command in the PowerCell. Now whenever the front end would make a request it would need to be the port used for the API, port 8080 in my case followed by one of the requests specified in the urls file.

```
[05/May/2020 17:22:22] "OPTIONS /api/products/1 HTTP/1.1" 200 0
[05/May/2020 17:22:22] "PUT /api/products/1 HTTP/1.1" 200 263
[05/May/2020 17:22:22] "POST /api/basket HTTP/1.1" 201 181
[05/May/2020 17:22:23] "GET /api/products/1 HTTP/1.1" 200 263
[05/May/2020 17:22:25] "GET /api/basket HTTP/1.1" 200 183
[05/May/2020 17:40:04] "GET /api/basket HTTP/1.1" 200 183
(env) PS D:\FYP\Application\env\MusicSite> python manage.py runserver 8080
```

Further details on how the back end was developed can be found in Appendix 10.2.3

Application Integration– 5.6

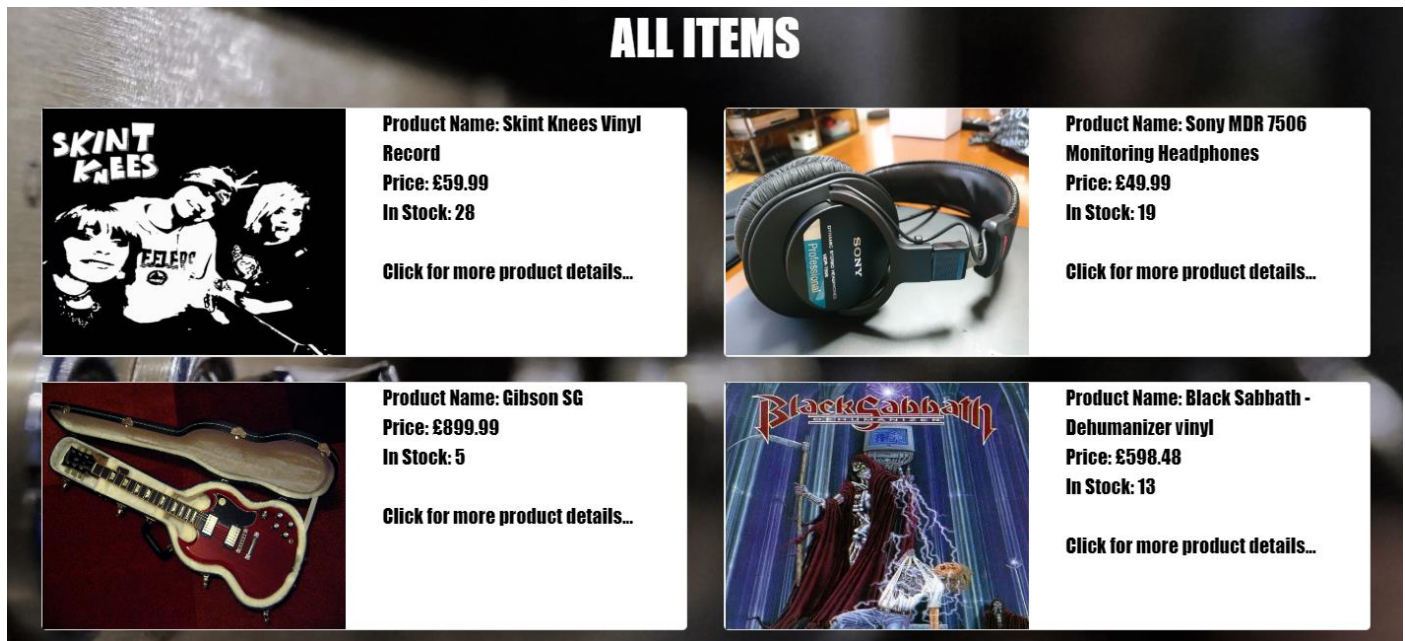
The final part of development now that the front end, the back end RESTful API and the database was set up was integrating all the parts to create the final project. This section of the project was definitely the more tedious and picky part of the development due to all the specific variable names, functionality and minor errors that weren't clear on how to solve.

To complete the integration of this application I would have to use Axios Promise based HTTP client for Node.js. This needed to be installed like previous packages. This is what I used to create the requests from the front end to get data from the API. Each of these requests would be different and would look like the ones created in the API files from the back end. This became frustrating at time as more than often I would accidentally use the wrong casing for the request which would create a error 500 as the request didn't exist so to fix this I would have to load the URLs from the API to make sure I spell checked the right requests to avoid this error. This was all done inside a componentDidMount function so that the request would only fire when the page loaded.

```
componentDidMount(){
    // when t
    Axios.get('http://localhost:8080/api/products/equipment').then(res => {
        this.setState({allitems: res.data});
        console.log(res); //inputs all data in
    }).catch((err) => {
        console.log(err);
    })
}
```

Another problem I encountered while developing this was a CORS error that I thought was already handled in the API. The error was caused by me originally whitelisting the Django server and not the React server, so the requests weren't allowed to be passed. This information I found was not clear on which had to be whitelisted and in turn was solved through trial and error.

After the data was successfully pulled in the database this had to be added to the right pages so that it could generate the cards specified earlier in the report. To do this I would have to loop through each of the JSON objects pulled from the API and create a new card for each of the items. This proved difficult as I would have to create the HTML for the cards before the render of the page. This way it was then able to load the HTML for each item that is pulled from the API and create a new card. The main problem with this was the again the naming of the variable. When pulling the data from the props the variable names had to be saved into a state first with the right names as well as then being called inside the HTML using the { } with the right names so that the correct information was being used. One minor miss spelling could prevent the data from reaching where it should. This was completed for all card-based pages including the basket as this works in a similar way just with another database.



The item page was easier as this would only pull one item at a time. Once the ID was passed into the link for the Item card then it was easy to use this to request specific items. This allowed me to find very specific items in the products database and fill out a details page using the props set from the initial API request. The primary issue with this, that did also show up with the cards was the images being sued. These images couldn't be saved to the database so instead they had to be saved locally and imported at the start of the page so they could be used depending on the product being used. This became problematic as not only did this slow the application down a little but if they weren't imported and required before everything else the image didn't appear at all. So, to have the image appear I had to sacrifice a part of the applications speed.


```
import NumberFormat from 'react-number-format';
let images = {
  item1: require('../imgs/item1.jpg'),
  item2: require('../imgs/item2.jpg'),
  item3: require('../imgs/item3.jpg'),
  item4: require('../imgs/item4.png'),
  item5: require('../imgs/item5.jpg'),
  item6: require('../imgs/item6.jpg'),
  item7: require('../imgs/item7.jpg'),
  item8: require('../imgs/item8.jpg'),
  item9: require('../imgs/item9.jpg'),
  item10: require('../imgs/item10.jpg'),
  item11: require('../imgs/item11.png'),
  item12: require('../imgs/item12.jpg'),
  item13: require('../imgs/item13.jpg'),
  item14: require('../imgs/item14.jpg'),
}
```

The more difficult part of this was posting data to the API through the checkout page. When completing the initial upload, the API sent back bad request in the console log of chrome as there was an error with some of the data being sent but no clear way to identify which. This was becoming a problem as then I had to search through the request in the front end as well as the API on the back end to locate the problem with the data. After searching these files, it was made clear that I had used an incorrect data type on the model of purchases so the validation wouldn't allow it to pass. Once these simple mixes up errors were fixed the application would finally allow the upload of data to the purchases table as well as the clearing of the basket table, which had to be emptied each time the app was completed.

After this the only real task was cleaning up the application and making it look better. This was straight forward, as this was only minor things that needed changing such as colour schemes in places, pages that clearly identify that a task has been successfully completed as well as a simple addition function on the basket page so that the user could see how much they had spent.

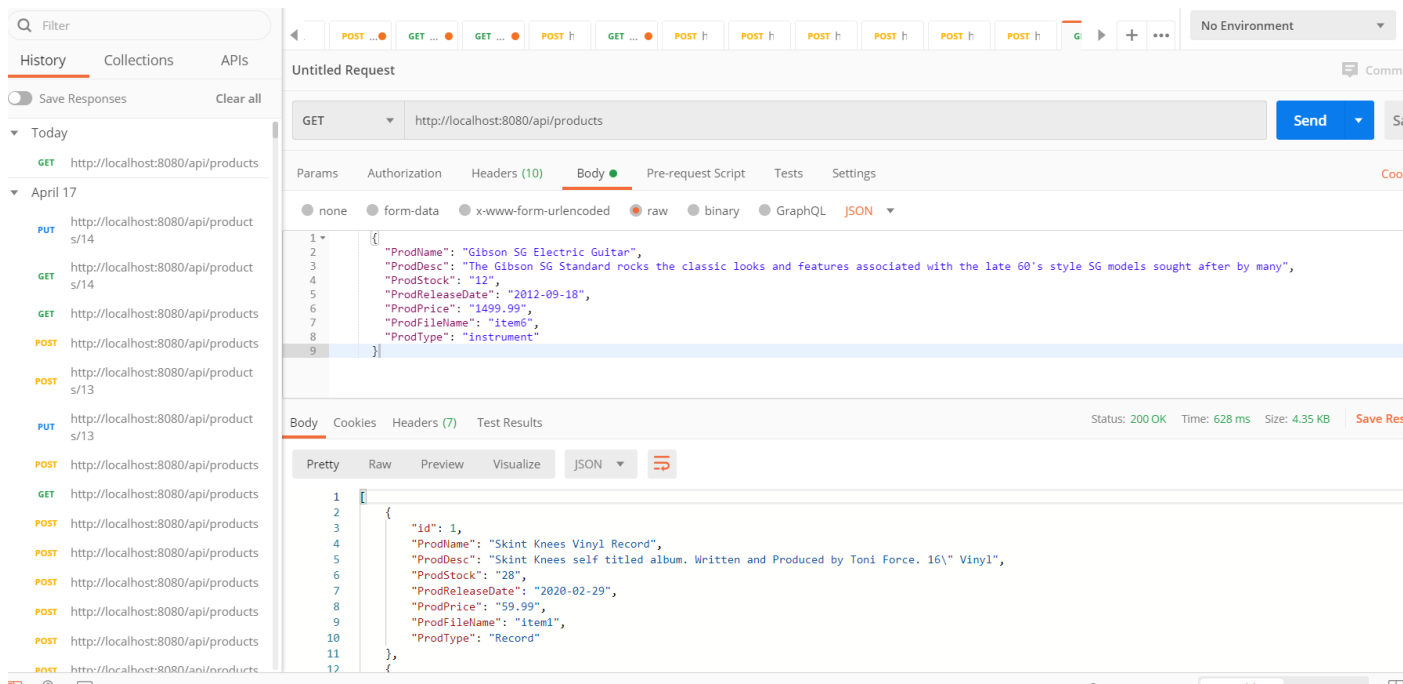
The addition function was slightly more difficult than I expected as when completing this addition, it would instead of adding the total together it put the numbers side to side with no clear reason as to why. This was an error of my own creation because of the data type the json was bringing back. When the data came back as JSON instead of saving it as a float or a double variable, a variable where addition can be used it was coming back as a string and so adding it to others were being treated as a string. To fix this, when the data came back, I would convert it to a float then complete the addition. This in turn then worked and allowed a proper, validated, price to appear at the bottom next to the checkout button.

```
componentDidMount(){
  Axios.get('http://localhost:8080/api/basket').then(res => {
    var gTotal = 0;
    res.data.map((rowData,i) => {
      gTotal += parseFloat(rowData.BskItemPriceTotal);
      console.log(gTotal);
    })
    gTotal = gTotal.toFixed(2);
    this.setState({allitems: res.data,
      total: gTotal});
    console.log(res); //input
    console.log(this.state.total);
  }).catch((err) => {
    console.log(err);
  })
}
```


Testing – 6

API Endpoint Testing – 6.1

End point testing is what is says on the tin. This would be the general testing of the end points for the API created during the development. To do this I could have used Unit testing, but this requires data to already exist as well as the data to not change as the test would have to be changed if this was the case. Instead I opted to use the software 'Postman'. This is an API based testing tool to make sure data can be sent, received and edited using the API requests outside of the application environment. This is a simple tool to use and to test with.



This software allowed me to enter all the possible requests into the bar at the top next to send and change the type of request using the drop down on the left to such things as GET, PUT, DELETE etc.

Using this application, I was successfully able to test all possible requests to the API as you can see a list of previous tests down the left side and across the top in tabs. From here I was first able to check that data was able to be collected from the API as a bulk using no parameters, using an ID parameter (file type) and using the ID of a item. If the results showed something similar to the bottom of the screen shot the request was a success. Otherwise I would throw back and error stating where the error originated from or if it was just a bad request.

After GET was finished I tried using POST to upload some test data that won't be displayed to check if the data was reaching the database. If this was a success it would send a message saying the data has been successfully submitted, else it would show the HTTP error code or what variable was prevent it from sending due to validation.

Once the POST testing was successful all that was left would be PUT for edits and DELETE to delete data. This again was used with test data so not to damage existing data. PUT would first require me finding the individual test object. Then from here I would be able to copy the data from the results of the GET into the body and edit some of the information saved. If when the PUT request was sent a message displayed saying edit was successful, the edit would have worked. Else there will have been an error with the new data entered such as text in a number field of a number in the date field etc. Once this was complete

DELETE was straight forward. It again would use the ID of an item but this time the method being used would be DELETE. This was a simple test as once the request was done, if the console said the item has been deleted, we could then go look at the table in Mongo Compass to check If it has been deleted.

All tests listed were a success as the right information was displayed after several attempts as well as the validation being properly handled and used when this was being tested also. The API could be properly implemented and properly used alongside the front end application.

User Testing – 6.2

For user testing I asked a select number of individuals who will remain anonymous to complete a series of tasks on both Amazon.co.uk and then again on my 'Future Of Sound' e-commerce application to work out which of the applications were more user friendly, faster, more efficient, easier on the eyes and overall performance on both to work out which would be more beneficial for an e-commerce site to use.

This testing was also to measure the success of the project as from their actions I will be able to see if the application has met all the necessary requirements.

NOTE: All user testing had to be completed over skype due to Covid-19. Covid-19 did not allow people that weren't in the same household to meet outside in a social manner and so face-to-face testing could not be complete. However, the testing could be completed over skype if the user had access to the internet to use both Amazon and be able to communicate instructions to me to complete on the application to simulate their use in what they might do. This does have the potential to compromise the results so once complete seconds may be added or deducted for the communication of instructions.

The questions and tasks can be found within the appendix. Here are their results/feedback

NOTE: Questions with a ']' contain answers for Amazon.co.uk and My Application. They are split in that order

Questions:

Users	Question 1	Question 2	Question 3	Question 4	Question 5
1	Amazon	My App	My App	My App	Choose Supplier Delete
2	My App	My App	My App	My App	Finding a product category Delete Screen
3	Amazon	Amazon	My App	My App	Finding Item Initial Load
4	My App	Amazon	My App	Amazon	Nothing adding an item to the basket
5	My App	My App	My App	Amazon	Item loading Deleting an item

Users	Question 6	Question 7	Question 8	Question 9	Question 10
1	My App	Easy Layout / Categories products	My App	Yes	Less submission screens
2	My App	Simpler	My App	Yes	Font
3	Amazon	Clearer on what can be done	My App	Yes	A profile/wish list feature
4	My App	Colour scheme less hard on eyes	My App	Yes	Remove success screens in between submissions
5	My App	The confirmation of the delete item	My App	Yes	Item reviews like Amazon, Add profiles

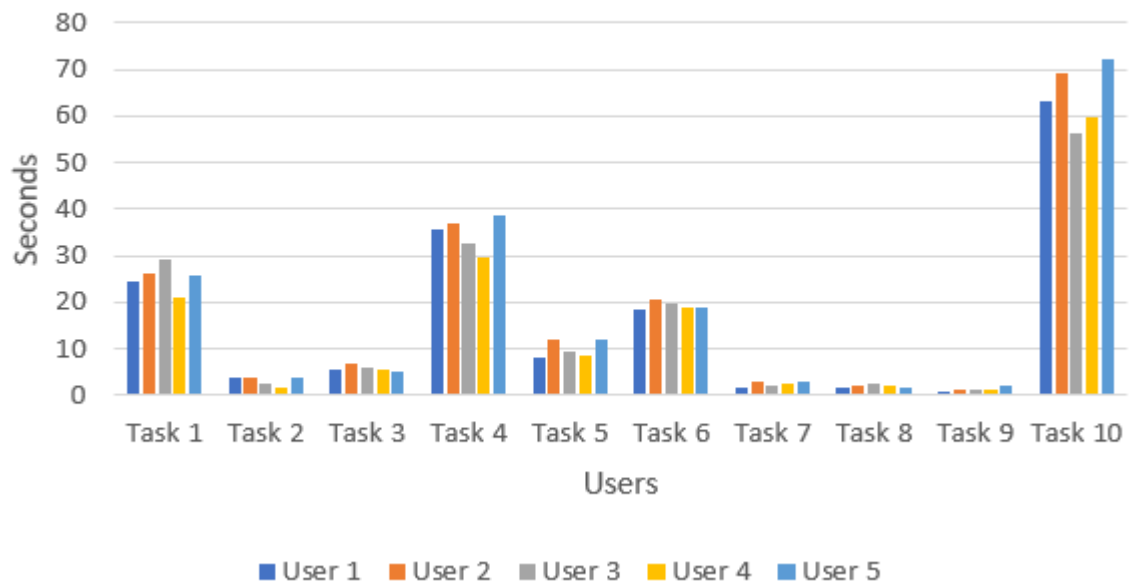
Users	Question 11	Question 12	Question 13	Question 14	Question 15
1	8 7	5 6	3 5	9 9	10 10
2	7 6	6 6	4 6	10 8	8 9
3	7 7	7 8	7 9	8 9	7 9
4	8 9	7 6	8 8	9 8	7 10
5	9 9	5 7	7 6	9 8	8 9

Tasks – Timed(seconds)

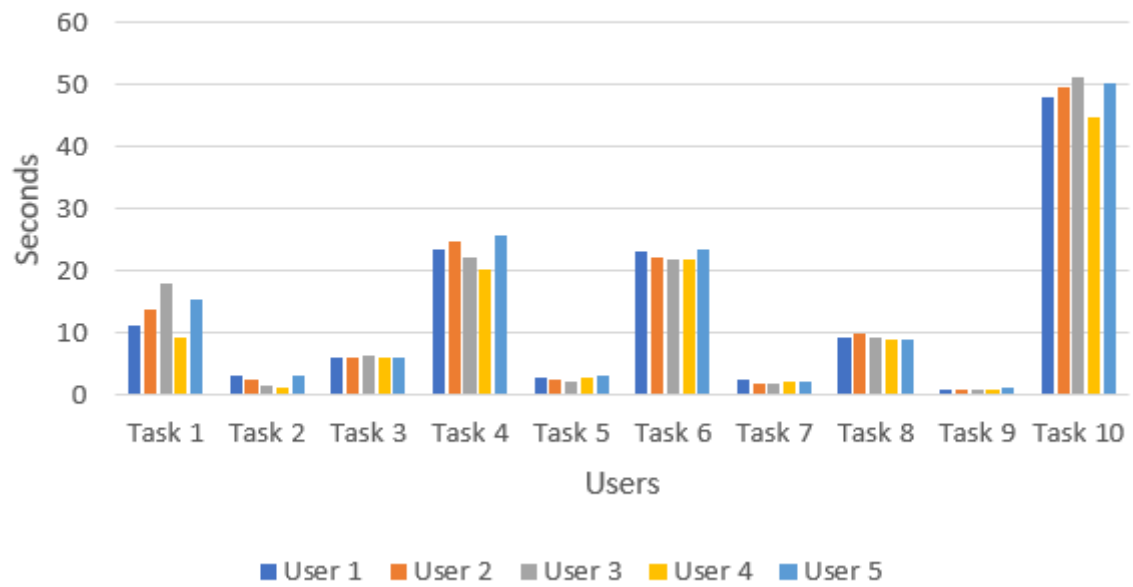
Users	Task 1	Task 2	Task 3	Task 4	Task 5
1	24.4 – 11.2	4 – 3.2	5.69 – 6.15	35.66 – 23.34	8.09 – 3.01
2	26.1 – 13.8	3.8 – 2.6	6.81 – 6.20	36.82 – 24.67	12.02 – 2.67
3	29.2 – 18	2.6 – 1.7	6.02 – 6.43	32.43 – 22.21	9.45 – 2.21
4	21.2 – 9.4	1.8 – 1.3	5.43 – 6.02	29.49 – 20.33	8.67 – 2.96
5	25.9 – 15.3	3.6 – 3.2	5.31 – 6.09	38.7 – 25.78	11.78 – 3.16

Users	Task 6	Task 7	Task 8	Task 9	Task 10
1	18.29 – 23.12	1.82 – 2.49	1.45 – 9.47	1 – 0.86	1:02.97 – 47.89
2	20.45 – 22.23	2.74 – 2.02	2.16 – 10.07	1.22 – 0.79	1:09.32 – 49.53
3	19.82 – 22.04	1.92 – 2.03	2.34 – 9.23	1.32 – 0.97	56.45 – 51.38
4	18.67 – 21.92	2.65 – 2.12	2.02 – 9.12	1.17 – 0.89	59.68 – 44.77
5	19.06 – 23.45	2.86 – 2.29	1.78 – 8.98	2.01 – 1.12	1:12.02 – 50.31

Timed Tasks - Amazon



Timed Tasks - Future Of Sound



Evaluation – 7

For this chapter of the report I will be evaluating the successes and failures of the 'Future of Sound' Single Page application and the overall successes and failures of the project. After this this chapter will contain any issues or challenges while developing this project as any ethical issues that may have risen. Finally, I will be going through all the personal and professional skills learnt while working on this project.

Evaluation of Deliverable – 7.1

Deliverable design - 7.1.1

The design and planning on this deliverable were very important to the success of the application as the choices made at this point in the development could determine how the end results of the application looks and works. The success of this can be measured during the testing phase with User testing.

First of all, the designs had to be completed on time to allow the rest of the applications development to fall in line behind it so to make sure this was complete the designs had to be complete before the start date of the front end development on January 17th 2020. This was a success as the quick sketches and requirements built for this application were completed in the week leading up to the 17th and proved very useful when building the front end. Very little changes had to be made.

The designs also had to meet a standard for the user as well as with users they will be looking for a handle full of things. Easy of use, how soft it is on the eyes, how clean the design is and its overall appeal. Looking at the results from the user testing in the previous chapter, this part of the design turned into a huge success as most users thought it looked more appealing and easier to use than that of Amazon.

The designs themselves that were on paper weren't the cleanest designs for the application but they were able to easily represent what the deliverable was supposed to look like and because some of them the designs lacked smaller details and were more aimed at the larger picture it allowed by fluid and adaptable small changes to the front end of the application overall.

Deliverable Development - 7.1.2

The main aim of this project was to create a single page application that is capable of completing similar tasks to that of a multi-page application to try and rival its speed to be able to evaluate them against each other. Looking back at the deliverable this was more than successful as the final product of the deliverable was in fact a single page application with a very similar range of functionality to that of 'Amazon'. Not only this its speed was able to rival that of 'Amazon' as you can clearly tell from the graphs on page 50 at the bottom of user testing. This graph clearly shows that on average the deliverable application was faster than that of 'Amazon' and so was easily evaluated against each other to see which would be faster and more efficient for the user in the short and long term.

Another main part of this application was to make sure that the application was easy to use for users of all skill sets. The results on page 50 and appendix 10.2.1 show that 100% of the users used for evaluating the usability and easy of use of the product was successful as not only did they find it easy but they found it easier than 'Amazon'.

The agile development process, Kanban, proved to be successful as well as not only was this properly applied by only completing a certain number of tasks each day during development but it also allowed me to complete the development before the deadline of the project. However, this did not reach the deadline originally set for late February due to external commitments that required my more urgent attention. Not

only this but a number of external factors such as strikes, and the outbreak of COVID-19 did delay the completion of the development as I gained external responsibilities that before were not an issue. The project was successfully finished but not on time.

The Kanban approach was not monitored on any external application and was just notes in a note book about what parts needed to be completed by a certain time.

When it came to the development I had some experience with JavaScript but not a lot as prior experience with web application and JavaScript was primarily through the use of PHP and about the backend I had no prior knowledge of any kind with programming using Python. To allow for a smooth development during my research I decided to practice using both frameworks so that, so I had a better grasp of how these frameworks worked. This way not only could I minimise catastrophic errors when the development of the project started but I also allowed me to understand the basic features of the frameworks so I had the ability to create very simple applications before creating a more advanced one such as my application.

Even though prior knowledge was learnt before development, there was still some dynamic learning happening at the same time as a lot of the application was unique and was not encountered during the practice phase. So small errors would still appear that would need researching before they could be properly handled.

Deliverable Testing - 7.1.3

Testing for this application was not completed all at once and was in fact completed at a staggered stage due to what was being tested and when it needed to be tested. Such as the back end API tests to check the availability if the end points was completed during the development of the API to make sure each request could be completed successfully as well as at the end to make sure that they were all able to be used in the order in which they will be used by the front end.

This made checking the functionality of the application easier as this way, before the function was added to the front end I could make sure that the API and application as a whole was capable of completing the requirements set at the start of the development. These were tested with a range of data to make sure than the validation added to each of the models would work as they should, so I was able to fine tune the models and functionality. This proved useful because not only was it able to show any model errors through the development and clearly work out the functionality for the user testing, it also was able to display any errors when connecting it with the front end if the back end was already working. So, if an error appeared and it was unknown by where it was coming from it would be from the front end.

User testing was completed by a range of people with different skill sets and was also used on this application for several reasons. The primary reason for user testing was to check its basic requirements as a lot of the requirements were heavily user based such as easy of use, clear to use, being able to add and delete items to a basket etc. This was tested using a series of tasks for the user to complete to make sure the requirements were met. If there was an error, I would know I haven't met all the requirements and then further development would be required. But looking at the results from the task with how quickly they were completed without any issues this was not the case and so all user requirements were met.

User testing was also in place so that I could collect results for both Amazon and my application to compare them against each other, due to the whole project being based around the speed and efficiency of each application. This was very useful as not only did the results show a clear that my application was much faster and more efficient, but it was able to show where my applications strengths and weaknesses were. But from these timed results for all users placed in a graph I was clearly able to see that a single page application would be much faster and efficient. Not only this but it showed how smaller e-commerce sites can benefit from single page applications and their speed compared to those of Amazon.

Overall the testing clearly shows that the application not only meets all the requirements and aims of the application but was able to work as it was expected to as well as how clear the UI of the application was for the user looking at results from the user testing. From these results it can clearly show where future development could reside in this application to improve its speed, general usability and functionality.

Project Evaluation – 7.2

There are a number of ways the success of this application can be measured as this application has a lot of components that can be measured but looking back at the overall question of this application. Which is the faster and more efficient type of e-commerce application for the users and overall. Comparing Amazon, a multipage application, to the 'Future of Sound' application, a single page application support by a RESTful API.

Looking at the test results from the user testing on pages 49 to 51, it is clear that my application compared to the single-page app is faster and more efficient than that of the multi-page app. As you can see from the time results on page 50 and 51 as well as the graphs to accompany them it is clear that that tasks given to them were much easier to complete as much clearer on how to complete than the multi-page. A few of the tasks were slow than the multi-page and these will be part of the future development added further down as the feedback from the users has provided information on how to deal with this.

Looking at the results to the questions as well 80% of the users thought that my application was much easier to use as well as much clearer to use than that of amazon allowing them to understand the application much faster and complete the tasks that were given to them a lot faster as well. Not only this but 100% of the users said they liked using the application and would use it again if this application was published and would become more commonly usable instead of in a testing environment. It was clear on how to add products and well as finding them though a vast collection of data.

Though the completion of testing it was also clear that all the user requirements were met as to complete all of these you would need to be able to complete all of the tasks available. With how easy users found the application to use as well as how easy it was to complete tasks, I believe this to be a success.

One thing was common though is that Amazon allowed the editing of items in the basket as well as the delete whereas my application only allowed delete. This could be changed in future development as part of other work.

Even though there were a few changes that could have been made looking back to make the application a little more advanced and more future proof overall I believe this project to be a success as the project was out to prove what type of web application would be faster and more efficient for the user and that was answered with the help of the requirements and the user testing at the end.

Issues & Challenges – 7.3

The project as a whole was a huge challenge as I had very little experience in terms of web JavaScript knowledge as well as no experience whatsoever in terms of Python so learning this language from very little to nothing was a challenge. Because of this there was a large learning curve on now only learning how to complete basic functionality with both of these languages but then to understand how to use this to work on the desired frameworks.

Another major challenge when working on this project was working on this and other assignments during strikes and the COVID-19 outbreak. During these times it was not as easy to access the right people and resources to be able to complete parts of the project, mainly through COVID-19. This was due to the limited access we were allowed to not only the university buildings but outside in general. These restrictions were strictly enforced by governing bodies as well as the University itself, so some notes,

information and other information was difficult to access. Not only this but it also affected how other assignments worked which means I had to apply more time to these, so they were able to meet the new guidelines and standards set because of COVID-19.

The primary issue was brought to my attention late on in the application how images would need to be implemented because of how I had create the database and API the application could not bring images from a storage area and had to be saved locally and brought forward when their file name was called from the database instead. This compromised the speed of the application as all images had to be imported for it to work. If given the chance I would apply more work into finding out how this could be done dynamically from the database instead if using data sent from the API to determine which image was able too load. This will cut down the load time of the application as well as increase the apps overall speed as there will be no need for any images being imported other than the ones that are present at all times.

This also caused testing to become a issue as originally the testing would be done face to face so that I could see the applications tested in action and how the user is reacting to them for more measurable results. Sadly, due to COVID-19 and its lockdown face to face testing was no longer an option due to social distancing so all testing had to be conducted over Skype. This could have possibly compromised the test results for both Amazon and Future of Sound due to the fact to complete tasks the users would have to look at my screen through screen sharing and give me instructions on what to do next. This would take time compared to what originally was planned but this was the only option left.

All other minor issues can be found throughout the development chapter of the report. These issues and challenges are a lot smaller than the ones listed above but still, at the time, proved to be an issue at times and at other times more of a challenge.

Ethical Issues - 7.4

When developing this application there were a number of ethical concerns to take into account as due to the nature of the application as well as the method of testing used at the end.

This application is an e-commerce site and so to test would require personal information to correctly complete a transaction such as name, address, email, bank details etc. Ethically this was not allowed to be stored as this information is personal to the user using the application at the time and if used incorrectly or if someone was to access the database then this user information I have stored would be at risk of being used without the person permission. This could have been handled in 1 of 2 ways to avoid ethical problems. The first way of resolving this could be that if their personal information is being saved, to let them know prior to the testing what the testing will include, what data may be being used and what I will be doing with this information such as storing it locally or on a external server etc. Then I would only use users who have signed the paperwork agreeing what they must do and what is being done with their data. But due to COVID-19 this will not be the case as the users cannot access the application to enter their data. Instead I opted to use the other method of, letting the users know what the user testing will be used for and instead of using their data, giving the user a set of mock details to work with. This way no one's personal data is being stored and potentially being put at risk of being used by a third party. This will also keep the users of the application anonymous if they didn't want any of their information being used at all. Users names were not recorded and instead stored as Users1 through to 5. This way if this project is published the user's names would be published along with it, potentially giving up their information and/or details.

Another Ethical concern I could not avoid is the transactions is the transaction itself. To properly set up a transaction I would need peoples card details and security information, which is very private to each user as this information can be used for criminal actions, so a functioning physical transaction using bank accounts and banking details could not be used and would instead have to be mocked using on screen

responses instead and store mocked transactions into a database. The transaction would not be seen by the user either way so it's not something they would be able to measure when using the application anyway.

One of the primary ethical concerns was the collection of the record covers and item images. All items on the application would need to have no copyright involved or a certain level of copy right that would allow me to use them as the product they are and not pass them off as another make believe product. For 1 of the images, the skint knees album cover, I was able to contact the band as well as the graphic designer behind the designs and album to ask for their permission to use this on my application. They agreed to it as this not only allowed them to show off their album to people who may not have heard from them but also because the copy right they have on the item allows it to be used as the product it is. The rest of the items and album covers were collected using google images advanced search to look for images that have 'free use or share, even commercially'. This way the images found on google images would only display images that have the right copy right around them for the use I need without using copy righted images. Most albums were old because they exceeded the copy right life span and have become free to use.

Skill Development – 7.5

Personal Skill Development – 7.5.1

The first personal skill developed in this project would be time management. With the amount of work needed to complete this with all the research involved, then design, development, testing and the writing of a report of this scale requires a lot of time management so that it isn't being rushed. To complete this I broke up all sections of the project up into smaller, more personal deadlines so that the project is completed on time. This allowed me to spread out the workload over the whole time it was available allowing me to manage my time with it around other external responsibilities such as family, other deadlines and over external factors that cannot be predicted such as strikes of COVID-19. You can see these mini aims in my specification

This was a very valuable skill as it allowed me to complete the application and report before the extended deadline even amidst all the chaos of the COVID-19 outbreak without rushing the project as a whole. Not only this but in the future, it will allow me to use this skill on other projects so that they are completed to the best of my ability, completed in the time frame given and not rushed.

Another personal skill developed is my research capabilities. Before this skill consisted of looking at 2 websites max and working off the information given from sources very similar to each other. After completing this I have found that there is much more information, useful sources and aids out there than previously. Now I understand that not only are sources available from sites but books and old papers as well allowing for a much more in-depth research scheme which will broaden my range of knowledge on a topic. This also will allow me to cover more information and understand more of a topic than just what it does but how it works as well as why it works that way. An example of this is the research chapter of the project talking about API as this uses a range of different web pages but also a book on API found online as well as data from a research paper. This allowed me to further my knowledge in what I would be developing with the API.

It would have been a useful skill to keep a diary of what was completed on what day and any problems encountered along the way as at the moment I only kept very basic notes which said what the error was and what changed about it to be fixed but I did not include all errors as some I tried to fix almost straight away and forgot to note and others lacked the context of the error and so was difficult trying to work it into the report.

Technical wise the I have not only broadened by skill set with the language JavaScript but I have learnt 3 others technical languages. The JavaScript framework React.js, basic python and the python framework Django. These languages were not known to me at the start of the project and were daunting at the start knowing how much practise these could have taken. Learning these languages has now increased my programming knowledge as well as increased my career opportunities knowing that I have multiple languages and frameworks under my belt. Not only this but it has also increased my knowledge into web application development from just multi page applications to single page applications and given me a deeper understanding of APIs when developing these applications too.

Professional Skill Development – 7.5.2

The first professional skill developed while working on this application is the ability to coordinate with others, or in my case with my supervisor. A steady flow of coordination was required between me and my supervisor for a number of reasons. For example I had to keep my supervisor informed on where I was up to, issues encountered during development, queries about the project and so we both understood what I was aiming to complete. Communication is key when working with others or for others as you may think one thing is right and work towards it and find out that it wasn't. communication was a little slow at the start but towards the end of the project communication towards to supervisor was increased as we were able to talk about issues such as how testing was going to be completed in lockdown of COVID-19 as well as any advice for the project such as what style of writing it should be completed and how it will be shown to him later on. This way we were always on the same page with the project and was able to help when needed or questions needed answering or clearing up.

Another professional skill I have developed is problem solving. This skill was developed during the development of the application while using languages and frameworks unfamiliar to me. During its development I encountered a few errors and problems that I have never encountered before because of how new these languages were to me. These problems were solved through a mixture of research and knowledge from previous languages that I have used in other modules. From this knowledge I was able to adjust and solve these problems which in turn then also helped increase my knowledge of the frameworks and languages used. Most problems were solved in the end as the application managed to reach the requirement and aims of the project successfully.

My final skill I developed through the creating of this project was judgement and decision making. This was primarily to do with the users and ethics behind the application and what to do to avoid them and keep the users safe. With all the copyright and internet safety around users and the ethics of the application a lot of judgment and decisions had to be made to make sure everything was up to standard and no one was at any risk when using the application. Such as the decision to keep users completely anonymous when in user testing. The information needed to complete a transaction is very personal and if used in an incorrect manor could result in the crimes potentially being able to be committed. So, to keep the user safe from this I opted to give each user a set of fake information to use at transactions so that none of their card information is at risk if someone got access to my local database.

Another example is the judgement call to complete testing over skype instead of cutting it out. Due to COVID-19 face to face testing could not be completed during lockdown. So instead of going out during lockdown and risking potential contamination of others and myself or not getting any results due to having no test subjects I opted to ask people who had access to the internet to complete the testing over skype instead. This way testing could still be completed and able to get test results and no users were at risk of potential contamination.

Future Development Ideas – 8

User Advised Developments 8.1

Firstly, I need to take into account the additional features the users would have wanted had they been given the chance to add parts to the application. These changes that users wanted were a lot smaller than that of additional changes.

First of all, the design changes. 60% of the users just wanted simple changes in the design or design functionality. One user had required a change in font as they thought the font was quite aggressive for the screen and would have been a lot softer font such as Aharoni as this font still keeps the bold and strong feel as the font used on the application without seeming too big and bold that it seems aggressive. The other 2 in this 60% only wanted the screens between actions removed. This way it would allow users to add multiple items to the basket at once instead of having to go back through the successful add screen to get back to them. Same goes for the delete. The users would have been happy if the button that says delete just deleted the item instead of taking you to another screen to confirm it as this takes time for a simple task. Both of these could be easily done.

The other 2 applications wanted profiles implementing, like on amazon, so that they have the ability to save items to their basket and come back to it on a later date to edit it or add more to it if they didn't have the money now. This way it saves items in their basket and would also allow for a faster transaction as most of the data could be filled out automatically for them and if they become frequent shoppers it saves them having to fill out the same details over and over. This is a much more difficult task which would maybe include more tables but could eventually be completed so they could do just that.

Additional Development 8.2

Support Chat system – This would be a small support window in the bottom right that would allow the user to talk to a IT technician or IT support if an issue did arise while on the application. With how this application is developed it could be done as simple page apps with APIs allow for live data to be sent and received. This would only be implemented if the app was published as it would require a user to be there to support the input of the user.

3rd party payment option – Another development could be the use of 3rd party options to pay for things. This in turn would eliminate the use for profiles in the previous developments as they could save all their card information and general information for purchases on these 3rd party apps. All they would need to do to successfully complete the transaction then would be to sign in and the application should be able to complete the rest. This may be a little more difficult as not only would I have to work out how this API works again it would need to be published so that I could complete transactions in the real world but would make the application a lot faster than it is already and in turn supports my evaluation even more.

Email receipt – This is a simple development for the transactions once again. Only this may require another server. The idea behind it is that once a transaction is complete it would use the email provided to send an email to the email address to confirm the purchase as a digital receipt. This way the user would be able to see what they bought, how many they bought, the company they bought it from and how much the grand total came to. At the moment it is only displayed on a screen after the transaction is complete so having an external factor that users can check when then need could be beneficial for the overall usability for the users

Card Capture API – This feature is more for people using the application on a mobile. When the user comes to input their card details it can be tedious doing on a phone. This could be changed if I implemented the use of the google card scanner. This would then take a picture of the front of your card

and input all the numbers needed for the transaction off the front of the card into the right input boxes, minimising the amount of work the user would do. Again, I would have to understand how the API worked first and add extra security to the app as it will be taking a picture of the credit card but overall I think, if published, this API could be successful.

References – 9

For blueprint, 2020, Agile Development 101:

<https://www.blueprintsys.com/agile-development-101/agile-methodologies>

For Medium, Liz Parody, 2018, waterfall vs agile methodology in software development:

<https://medium.com/@lizparody/waterfall-vs-agile-methodology-in-software-development-1e19ef168cf6>

For Inflectra, 2020, Kanban:

<https://www.inflectra.com/methodologies/kanban.aspx>

For Amazonaws, 2020, Comparison between agile and traditional methodology: (Source no longer accessible)

https://s3.amazonaws.com/academia.edu.documents/58993716/10.1.1.464.609020190422-13963-j0ju8a.pdf?response-content-disposition=inline%3B%20filename%3DA_Comparison_between_Agile_and_Tradition.pdf&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=AKIAIWOWYYGZ2Y53UL3A%2F20200313%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-Date=20200313T152831Z&X-Amz-Expires=3600&X-Amz-SignedHeaders=host&X-Amz-Signature=a189e0c00665a16280560e6f613ec514d501d468758805c38d40e4f715e27240

For researchgate, Mario Špundak, 2014, Mixed Agile Traditional Project Management Methodology – reality or illusion:

https://www.researchgate.net/profile/Mario_Spundak/publication/270846766_Mixed_AgileTraditional_Project_Management_Methodology_-_Reality_or_Illusion/links/571d2bae08ae408367be552e/Mixed-Agile-Traditional-Project-Management-Methodology-Reality-or-Illusion.pdf

For HowToGeek, Chrishoffman, 2018, What is API:

<https://www.howtogeek.com/343877/what-is-an-api/>

For Lexico, Oxford, API:

<https://www.lexico.com/en/definition/api>

For Medium, Amanda Kothalawala, 2018, What is an API? How does it work?:

<https://medium.com/@ama.thanu/what-is-an-api-how-does-it-work-f4ea552d741f>

For Raygun, Anna Monus, 2020, SOAP vs REST vs JSON – a 2020 comparison

<https://raygun.com/blog/soap-vs-rest-vs-json/>

For Medium, Mari Eagar, 2017, What is the difference between decentralized and distributed systems?

<https://medium.com/distributed-economy/what-is-the-difference-between-decentralized-and-distributed-systems-f4190a5c6462>

For Semanticsholar, Anil Dudhe, Swati S. Sherekar, 2014, Performance Analysis of SOAP and RESTful Mobile Web Services in Cloud Environment

<https://www.semanticscholar.org/paper/Performance-Analysis-of-SOAP-and-RESTful-Mobile-Web-Dudhe-Sherekar/af703c355f4287cf4fa40c6b939b79934b25d2a4>

For Wavemaker, wmrnaresh, 2019, Page Concepts

<https://www.wavemaker.com/learn/app-development/ui-design/page-concepts/>

For rubygarage, Anastasia Z., 2018, PWhat's the difference between single-page and multi-page apps

<https://rubygarage.org/blog/single-page-app-vs-multi-page-app>

For Medium, Goldy Benedict Macquin, 2018, Single Page applications vs multi page applications – Do you really need an SPA?

<https://medium.com/@goldybenedict/single-page-applications-vs-multiple-page-applications-do-you-really-need-an-spa-cf60825232a3>

For Medium, Neoteric, 2016, Single-page applications vs Multiple-page applications

<https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58>

For Pro Single Page Applications Development, Gil Fink, Ido Flatow, SELA Group, 2014, Using Backbone.js and ASP.net

https://books.google.co.uk/books?hl=en&lr=&id=ayuLAWAAQBAJ&oi=fnd&pg=PP3&dq=+single+page+applications&ots=4fS4Fa5-5S&sig=QUTGTTiwlMYxTyr8eH0bp29E05E&redir_esc=y#v=onepage&q=single%20page%20applications&f=false

For NU:RO

<http://nuro.co/>

For Xbykygo.com

<https://www.xbykygo.com/uk/>

For Seedlipdrinks.co.uk

<https://seedlipdrinks.com/uk/>

For codeinwp, Shaumik Daityari, 2020, Angular vs React vs Vue: Which framework to choose 2020

<https://www.codeinwp.com/blog/angular-vs-vue-vs-react/#part-4-working-with-the-frameworks>

For data-flair, DataFlairTeam, 2019, Django Advantages and disadvantages – Why you should choose Django

<https://data-flair.training/blogs/django-advantages-and-disadvantages/>

For Clariontect, Rakteem Barooah, Applications that work best with NoSQL

<https://www.clariontech.com/blog/applications-that-work-best-with-nosql-database>

For Seeromega, James Burns, 2019, Django Vs Node.js: A Detailed Comparison, Pros and Cons

<https://seeromega.com/django-vs-node-js-comparison-pros-cons/>

For Guru99, 2020, SQL vs NoSQL

<https://www.guru99.com/sql-vs-nosql.html>

For uoa.eresearch.github 2014, Python Virtual Environments

<https://uoa-eresearch.github.io/eresearch-cookbook/recipe/2014/11/26/python-virtual-env/>

Appendices – 10

Questions/Tasks – 10.1

Questions

- 1) Which was easier to use in general?
- 2) Which do you think is the most responsive?
- 3) Which was clearer to use?
- 4) Taking into account your experience with the applications, which of these 2 applications do you think which do you believe was a better user experience in terms of how easy it was to purchase a product or collection of products?
- 5) Were there any parts of either application that seemed to take longer than it should have?
- 6) Which was a more visually appealing site
- 7) What about my application is better than Amazon?
- 8) Which was Easier to navigate
- 9) Are you Likely to use this site again, if it was used in the real world?
- 10) Are there any features or changes you would like to see being added to the application?

- 11) On a scale of 1 to 10 how quickly could you find a product on both applications?
- 12) On a scale of 1 to 10 how fast could you add an item to your basket?
- 13) On a scale of 1 to 10 how easy was it to complete a transaction?
- 14) On a scale of 1 to 10 how easy was it to edit your basket/delete an item?
- 15) On a scale of 1 to 10 how clear was both applications in terms of completing the tasks given to you

Tasks –

- 1)Search for [Item Name]
- 2)Find Product Information
- 3)Add [item] to basket
- 4)Find all items
- 5)Find Product Type Specific Items [Item Type Given during test]
- 6)Add multiple Items to basket
- 7)View basket
- 8)Delete Item From basket
- 9)Find Grand total
- 10)Complete Transaction

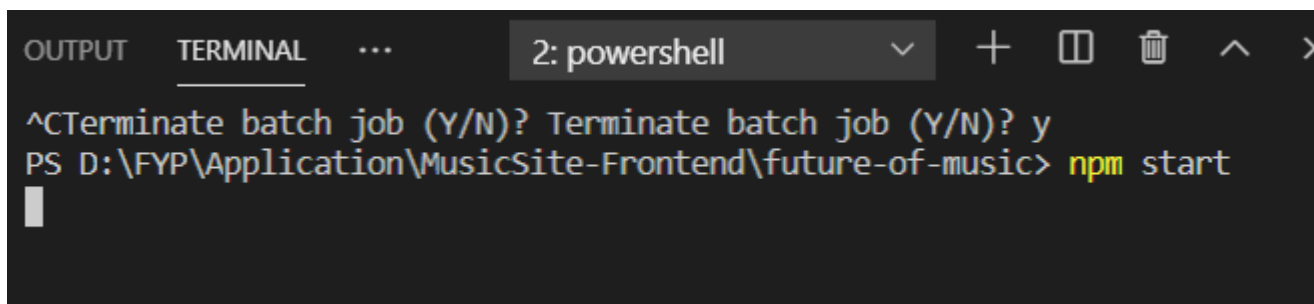
Development Steps – 10.2

Front end Development – 10.2.1

To start the development of the application I thought it best if first I built the front end of the application so that the general design is all set and ready for when it is connected to the API. Also, if I built the front end first it would be a lot quicker and easier to test if the data is making it to the front end of the application or not.

To start of with when using react you have to host it on a server based application for it to run so, I had to start of opening visual studio code, the programming software I used to create the application and direct the file explorer to the folder I will be storing the application inside. Once here the folder will become visible to the developer on the left side of Visual studio code 2019, to make it easier to navigate between files when development starts. Once this has been set the I right click on the file the app will be stored in and click open in terminal. At the bottom of the screen this will open up this folders directory in a Node.js command prompt.

Before we can start programming the React.js application a few things will need to be set in this command prompt. The start the React.js project you first have to create the application using a Node.js command. This command is 'npm init react-app (app name)' or in my case 'npm init react-app future-of-music'. This will start to set up all the file layout, initialising files and folders and set up the framework ready to be developed. To check this application has been set up properly, once the React app has finished building, I enter the command 'npm start'. This will then run the application hosted on the Nodes.js server and open the application in the selected browser stating that you have successfully set up a react app if it has worked. The entry point for this application is saved in index.js. This is where all the different pages are rendered through so that the webpage can change dynamically. Once I knew it worked I used the command Ctrl + C to stop the server ready to install more libraries

A screenshot of a PowerShell terminal window. The title bar shows '2: powershell'. The terminal output includes a prompt '^CTerminate batch job (Y/N)?' followed by 'y', and then 'PS D:\FYP\Application\MusicSite-Frontend\future-of-music> npm start'. A cursor is visible on the line following the command.

```
^CTerminate batch job (Y/N)? y
PS D:\FYP\Application\MusicSite-Frontend\future-of-music> npm start
```

Another install required before the development of the application could start is the React Router Dom. This is the tool I used to create links between pages on the screen. React doesn't have anything like this preinstalled in it so to create a feel of the webpage changing pages like a multi-page without creating multiple pages this needs to be installed. This way when text is added to the URL the Router Dom will know what it is supposed to do. For example if '/home' appears in the URL it will render the home screen or if '/all-items' is there it will render the all items page. The Command for this is 'npm install react-router-dom -S'. Make sure that you use capital 'S' at the end of this command otherwise the router will not install all the correct files. Without this the web app would not be able to have more than one page, so an e-commerce site would be difficult to develop.

Once this has been installed successfully you will need to import this to the main JavaScript file where it will render the pages dependent on the URL. Inside App.js the code 'import {BrowserRouter as Router, Route} from 'react-router-dom''. This will then allow for the use of the Router library in the application. The final thing I had to install before development could start is bootstrap. This is a simple CSS framework that makes designing and position items on a web page easier. For this we need the command 'npm install

react-bootstrap bootstrap' and again on App.js import it using 'bootstrap/dist/css/bootstrap.min.css'. Now the application is ready to be developed in the front end.

To start off I needed to create a header. The header is more of the more important components as this had to be displayed on each and every page available to the user. So in the folder src I created a folder called components. This is where we will store all of the different parts of the application that will be rendered later on. Then I created a file called header.js. Once inside this file the file had to be made into a class that extends Component. This could be done by writing a few lines of code or by using Visual Studios on board libraries. If the developer installs the React-Native snippets extension to Visual Studio Code the developer can create the basic structure of a component when needed. Once I installed this, I entered the characters 'rce' in the code and press enter on the first result. This will create a basic component like so.

```
import React, { Component } from 'react'
export class test extends Component {
  render() {
    return (
      <div>
      </div>
    )
  }
}
export default test
```

Once this is in place, I could start building the header. **NOTE:** The one above is an example and not one the components used.

Here is where the general design of the header can begin. I used some of the bootstrap components such as "col-md-2 to spread out all the links in the header so that they weren't all cramped together. This was contained inside a class called row so that they would all be displayed next to each other instead of on top of each other and this was all contained inside a bootstrap container class so this way the links don't spread too close to the edge of the screen creating a nice clean looking header. To create the links on this page, react-router-dom needed to be imported again as 'Link'. This is how links between pages will be generated instead of the traditional method for multipage apps where it's just another URL. Each Link has a to component in it that when pressed will change the directory in the URL above. This is what the react-dom records when rendering screens. It will check on what the latest 'to' was and change the screen depending on that.

Finally, for this header I introduced a Logo like some of the examples did. This was given it's own inline styling so that it fit in the correct spot on the header as it caused a few errors when contained in the container. It would mess up the styling of the nav bar as well as get stuck to one side of the header instead of being dead centre. This was included using the tag built into HMTI that will allowed me to include local files, such as this logo, into the page. This had to be done using a require though. Just importing it though the src didn't render the image properly as the image needed to be one of the first things to render

for it to appear. This is why `require` was used. This then renders the image first before the rest of the HTML around it allowing it to render properly.

The inline styling was added to a const variable at the bottom of the screen so instead of having a lot of styling in the tags making it look messy it could all be changed at the bottom and added using the name of the variable. Another small feature that was added to the App was the use of awesome fonts. This is a basic font-based library that is used to bring in small icons that can be used in the program. These were placed at the side of each link that represents where the link goes such as basket or equipment.

```
import React, { Component } from 'react';
import { Link } from 'react-router-dom';
import 'font-awesome/css/font-awesome.min.css';

export class Header extends Component {
  render() {
    return (
      <div style={headerStyle}>
        <div>
          <Link to="/home"><img src={require('./imgs/logo3.png')} style={{width: "8%}}></img></Link>
        </div>
        <div className="container">
          <div className="row">
            <div className="col-md-2"><Link to="/home" style={linkStyle}> Home <i className="fa fa-home"></i> </Link></div>
            <div className="col-md-2"><Link to="/all-items" style={linkStyle}> All Items <i className="fa fa-list-ul"></i> </Link></div>
            <div className="col-md-2"><Link to="/records" style={linkStyle}> Records <i className="fa fa-circle-o"></i></Link></div>
            <div className="col-md-2"><Link to="/instruments" style={linkStyle}> Instruments <i className="fa fa-music"></i></Link></div>
            <div className="col-md-2"><Link to="/equipment" style={linkStyle}> Equipment <i className="fa fa-headphones"></i></Link></div>
            <div className="col-md-2"><Link to="/basket" style={linkStyle}> Basket <i className="fa fa-shopping-basket"></i></Link></div>
          </div>
        </div>
      </div>
    );
  }
}

const headerStyle = { // styling for links
  background: '#000',
  color: '#fff',
  textAlign: 'center',
  padding: '10px',
}
```

There were a few problems with styling the logo at the top as some times it would render 1px wide and appear on the right hand side and other times it would push all the text around. This problem was fixed by creating a fix position and width of the image so that it couldn't stretch further than it needed it and with the correct positioning in `headerStyle` I was finally able to centre it.

Once the header was complete it had to be added to the `App.js` to make sure it rendered once and appeared on every page. This was done like so.

First the component had to be imported into `App.js` so the file knew what to render. It was also imported into a instance of that import so that it can be used in the HTML.

```
import Header from "../components/Header";
```

Then after this inside the return of the render function the `<Router></Router>` tags need to be added so the application knows what it will have to load and when it will be loaded. Once these are in the place, between the Router tags the instance of Header can be place, inside a set of it's own tags like so. `<Header />` Using the `'/'` inside of the initial tag will open and shut the tag without the need to writing the closing tag. Often used when nothing is needed to be passed in or contains all it needs to. Like so

```
<Router>
  <div style={{backgroundImage: `url(${ba
  <Header />
```

Because this isn't a Route Tag that contains a path needed to access it, it will load once the App.js file has loaded and will stay at the top of the screen the whole time the user is on the application. After this was

Now that the header was finished and loading on all pages, I needed to make sure the background that I planned to add for the background of each page would do the same. I could have done this by importing the image onto each of the individual pages and have it render each time a page is loaded but this could have compromised the speed of the application that I was trying to test. Not only this but it was very unnecessary so instead I planned to add the background styling to the App.js just before the header so that it covered the whole page before items began rendering over it. This proved tedious and frustrating with how the image would work. When first adding this background image to the app it would load all rendered screens below it instead of over the top of it and there was no clear reason why. I have imported the image like the logo so the image was one of the first things to render so that it would appear but the styling for the tag was messing with how it would have worked.

The first error when fixing this issue was the tags. I had opted to use <image> tags because it was an image. This was a minor error as I had the tag id wrong. It was supposed to be . Second of all these tags are for when you are placing a image over the top of something not behind everything so these tags wouldn't have worked either. Eventually I opted to use the standard <div> and applied the image using the css, backgroundImage property. This had to be done inside '{ }' as this is what is used in jsx files in react to change the state of something dynamically, so it would be able to render the imported background image into the property. The problem is that text was still appearing below it and not on top of it. This is when I found that to keep an image at the back I would have to fix the image into position using the 'position: fixed' css property for this div and change the container with all the different page information to relative. This way the image wouldn't be able to move but the text would be able to sit freely where it was able to.

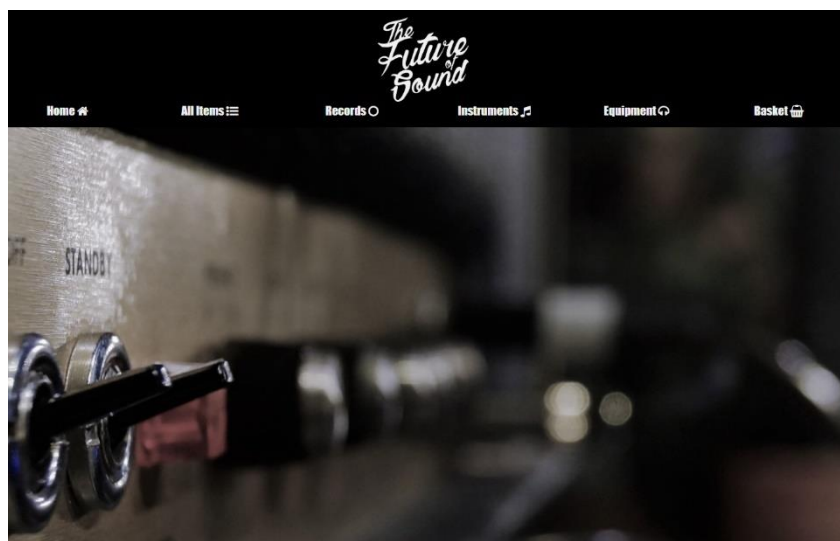
```
<Router>  
  <div style={{backgroundImage: `url(${backImg})`, maxWidth: "100%", minWidth: "100%", height: "100%", position: "fixed"}}/>  
  <Header />
```

The only other issue I had with the background image was it's sized. The standard size of the screen was smaller than the image so the make sure it fit perfectly without any overlap the images height and width had to be set the to 100% the size of the screen available. Once this was complete the background image was in place and ready for the rest of the pages to be developed.

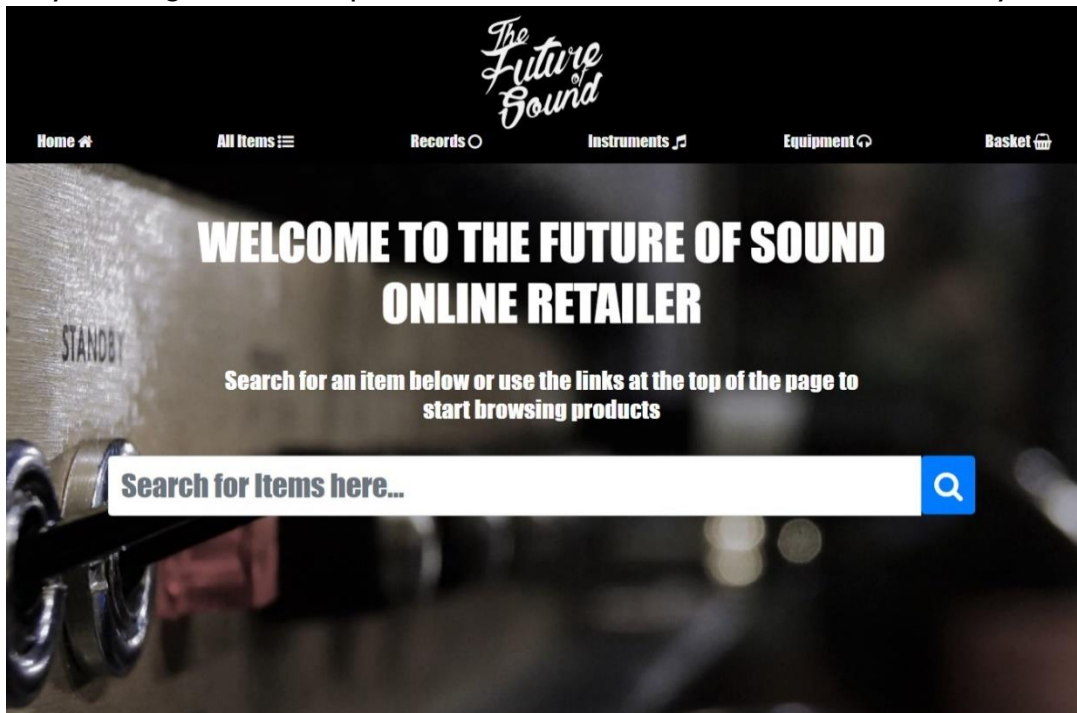
To start off with I thought developing a simple homepage would be the best place to start as this would be the first page that all the users would see.

Just like the header, the home page was created by creating a new .js file named home inside of the components folder. Then the component was build by using the visual studio extension 'rce' to render the basics of a component. Just

like the header, home will also need the import of react-router-dom for the links to other pages as well as awesome fonts for some of free icons available on it. This page was a simple page to develop as all it would contain would be a welcome message to the user and a search bar with a button at the end. This was easily developed using bootstrap and standard HTML to centralise all the information needed.



This was also the case for developing the checkout page for the application as this was just a large list of input boxes styled using the bootstrap classes with a submit button at the end. The only real difference



was that the checkout page was done inside of a `<form>` tag ready for submitting data to the database later on.

The main 3 pages that needed to be designed were the pages that hold the cards with the items on, all of which would be the same, the individual item page and the basket page. These would be generated for each item in a database so these would need to be built dynamically. For now, though they would be developed using raw HTML and Bootstrap to get a good idea on what they will look like before integration.

To start off with I build the item cards that will be used on 5 different pages, items, records, equipment, and search results pages. All these pages again were created in the same way as previous pages. All pages are started the same way. With a file being added to the components folder, and component basics being created using the `rce` command before adding the `react-router-dom` and awesome fonts.

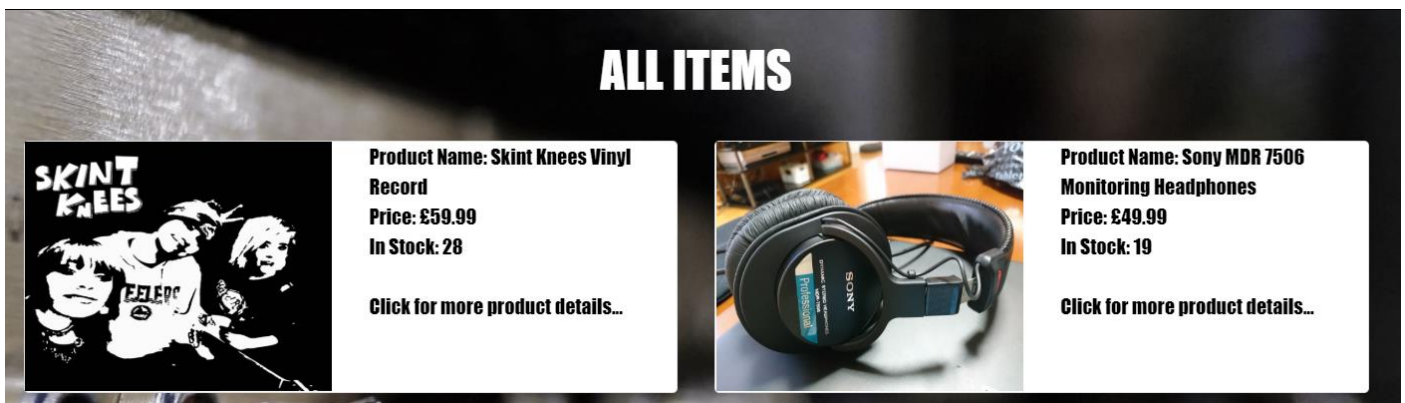
When building the card pages, it proved a little more frustrating than previously realised. I started out by setting up the page, all items, using a div with the `className` of `container` like I did the header as well as setting up the div with `className` `Row` just like the navigation options on the header too. This way every card I built would appear next to each other unless the row was too short, in which case it would appear below instead.

The problem with this is after looking up certain components of bootstrap I was using a div with the `className` `panel` after locating what looks to be a good visual plan for displaying the information. This became a problem as when run the application would display the mock data I had added to the panel but not the panel. I tried a few combinations of items to see if I could make it work, with very little success. After a while I looked up the bootstrap for a panel. The problem was my method turned out to be out dated code. Panel was the main apt of bootstrap v4. The version I had instead was the more recent v5 which changed the `className` from `panel` to `card`. Once the card started to show up I was able to style it. I styled it so that there were 2 items per row so that the item could see multiple items at once, that contain a small image if the product on the left and the name of the product and a little description next to it.

```

<div className="col-md-6" style={{paddingBottom: "20px", fontFamily: "impact"}}>
  <Link to={"item/" + props.item.id} style={linkStyle}>
    <div className="card" style={itemCard}>
      <div className="row">
        <div className="col-md-6" style={{overflow: 'hidden', height:"200px"}}>
          <img src={images[props.item.ProdFileName]} style={{height: "100%", width: "100%"}}></img>
        </div>
        <div className="col-md-6">
          Product Name: {props.item.ProdName} <br></br>
          Price: £{props.item.ProdPrice} <br></br>
          In Stock: {props.item.ProdStock}<br></br>
          <br></br>
          Click for more product details...
        </div>
      </div>
    </div>
  </Link>
</div>

```



The code above creates both of those cards seen above. The code will only generate 1 at a time at this point as it has been hard coded in for design purposes. The reason Link was imported to this page is so that I could turn the whole card into a link that will go to the individual item page. This page will then load the item selected into a page displaying a larger image and a more in-depth description of the item.

This code was added Records, Equipment, Instruments, All-Items and SearchResults as all of these pages would load the products and display them in the same way. These pages differ from what information they will receive later on.

NOTE: All pages being developed have all been imported like the header I the App.js file so the links at the top of the header are able to navigate to all these pages plus any links on these pages are then able to navigate to other pages such as a card linking to the Item page.


```

import Item from "../components/Products/Item";
import AllItem from "../components/Products/All-Items";
import searchResults from "../components/Products/searchResults";
import Records from "../components/Products/Records";
import Instruments from "../components/Products/Instruments";
import Equip from "../components/Products/Equipment";
import ItemAdd from "../components/Products/ItemAddConfirm";

import Confirm from "../components/Purchase/Confirm";
import Basket from "../components/Purchase/Basket";
import Checkout from "../components/Purchase/Checkout";
import ConfDelete from "../components/Purchase/ConfirmDelete";
import DeleteBasketItem from "../components/Purchase/DeleteItem";

```

```

<div className="container" style={{position: "relative"}}>
  <Route exact path="/" component={Home} />
  <Route path="/home" component={Home} />

  <Route path="/item/:id" component={Item} />
  <Route path="/all-items" component={AllItem} />
  <Route path="/search-results/:title" component={searchResults} />
  <Route path="/records" component={Records} />
  <Route path="/instruments" component={Instruments} />
  <Route path="/equipment" component={Equip} />
  <Route path="/add-item/:id" component={ItemAdd} />

  <Route path="/basket" component={Basket} />
  <Route path="/checkout" component={Checkout} />
  <Route path="/confirm" component={Confirm} />
  <Route path="/confirm-delete" component={ConfDelete} />
  <Route path="/delete-item/:id" component={DeleteBasketItem} />
</div>

```

These work in a very simple way thanks to the router dom. The Path is what will appear in the URL when a link navigates to when pressed. This path also identifies what needs to be displayed in the URL for the corresponding page or In this case imported component to be rendered. So if a link with the 'to' going to 'records' is pressed, this page will read that, find a page that has the path '/records' and load that page.

The individual item page was built using some of the knowledge of the previous card page. As the item page instead of containing multiply cards like the one displayed above the item page would consist of one of these. The idea being that the item ID would render in when the page loads and fill in the necessary parts with information from the database when connected so this is the only item page.

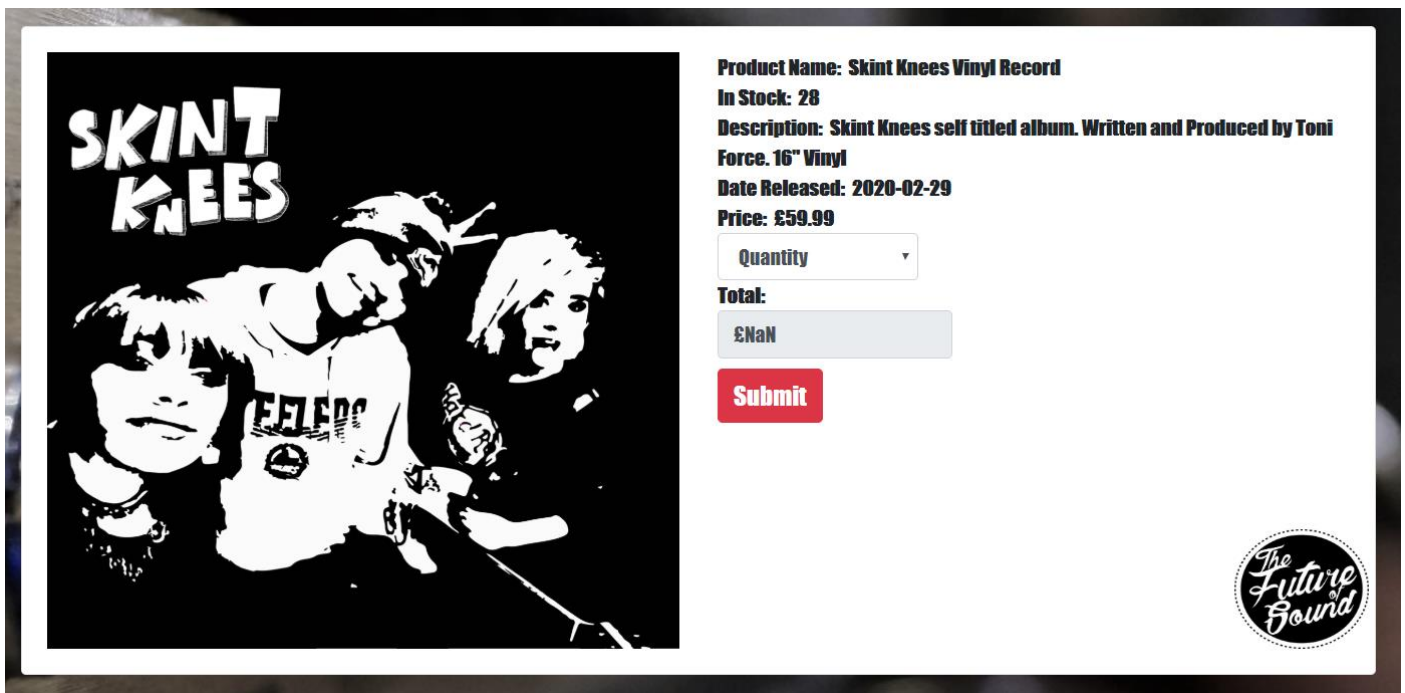
I built this page to look identical to the card only filled with more information. Again, built using primarily HTML and bootstrap classes. Although a little inline CSS was used to position all the parts of the page to the centre. They are built so that a image will render on the left filling half of the card using more column classes from bootstrap. E.g. col-md-6.

The Bootstrap columns consist of 12 equally broken up parts that can be put together into 1 part depending on the data entered. So, when I used col-md-6 it will use half of the card. This Will then be set to a specific size, for consistencies sake when different size images are used. These will never exceed the size of the user's screen. Then on the right side of the card I have added such items of data such as Name of the product, Description, date released, price who it was made buy etc. There has also been added a drop-down box with the numbers 1 to 5 in it and a submit button. I do this to limit the user to 5 items so later on they can't buy the entire stock. Plus it's part of the requirements stated earlier on.

This will be used later identify how many of the item the user wants before submitting the order to the basket. I have used the bootstrap styling so that everything matches, and it gives a more professional look to the inputs and buttons available. Alternatively, I could have designed them myself using CSS and including the style sheets like I did with the header style but this would have taken time to get right where as with bootstrap I could just add one className to a button or input and it would style itself. This could be used on all pages due to bootstrap being installed to the Node.js Server.

To allow this to be submitted to the basket later on the dropdown box and button at the end have to be stored inside <form> tags. Otherwise no data can leave the front end when submitting.

```
<form onSubmit={this.onSubmit}>
  <div className="card">
    <div className="card-body">
      <div className="row">
        <div className="col-md-6" style={{height: "475px"}}>
          <img src={{images[this.state.ProdFileName]}} style={{height: "100%", width: "100%"}}></img>
        </div>
        <div className="col-md-6" style={{height: "475px"}}>
          <strong>Product Name: &nbsp;{{this.state.ProdName}}</strong><br></br>
          <strong>In Stock: &nbsp;{{this.state.ProdStock}}</strong><br></br>
          <strong>Description: &nbsp;{{this.state.ProdDesc}}</strong><br></br>
          <strong>Date Released: &nbsp;{{this.state.ProdReleaseDate}}</strong><br></br>
          <strong>Price: &nbsp;£{{this.state.ProdPrice}}</strong><br></br>
          <select onChange={this.onChangeQuantity} required="true" className="form-control" style={{width: "160px"}}>
            <option value="-1" selected disabled>Quantity</option>
            <option value="1">1</option>
            <option value="2">2</option>
            <option value="3">3</option>
            <option value="4">4</option>
            <option value="5">5</option>
          </select>
          <label><strong>Total: <br></strong><input className="form-control" disabled value={"£" + parseFloat(this.
            <input type="Submit" className="btn btn-danger" style={{fontSize: "20px"}}/>
            <br></br>
            <img src={require('../imgs/logo.png')} style={{width: "20%", height:"20%", bottom: 0, right: 0, position: "ab
          </div>
        </div>
      </div>
    </div>
  </form>
```

All bolder text was Stored inside `` tags. This automatically makes all text inside the tags bold. Alternatively, I could have added inline styling to it bold but this tag is built into HTML and may as well be utilised as it doesn't require me having to change the styling of each area of text.

Finally, the development of the design of the basket. Building the basket was a combination of the item page and the pages with listed items on. The basket again uses cards but like the individual item page the card is the full width of the container in is sat in although a lot smaller in height as this page will have to contain multiple different cards at once. These cards would then contain the items image, Title, Quantity, individual price and total price. These were smaller cards so that the user could see more of what is in their basket at one time instead of seeing all the products information. These were broken up into smaller columns so more data could be visible too, so these were split up using `col-md-1`, `2` and `3s`.


Right at the bottom is some large text about total price and a check out button. The check out button is a link using the `<link>` tags again so that it can redirect to the checkout page mentioned earlier on and the total will be a generate total price of the order when data is brought in from the database. Most of this page again was developed using bootstraps cards and containers to keep the styling through the web application consistent and well-spaced.

```

<div className="col-md-12" style={{paddingBottom: "20px"}}>
  <div className="card" style={itemCard}>
    <div className="row" style={{paddingTop: "20px"}}>
      <div className="col-md-1">
        <img src={{images[props.item.BskItemFileName]}} style={{height: "100%", width: "100%"}}>
      </div>
      <div className="col-md-3">
        <strong>Product Name: <br></br>{props.item.BskItemName} </strong> <br></br>
      </div>
      <div className="col-md-1">
        <strong>Qty: <br></br>{props.item.BskItemQty}</strong> <br></br>
      </div>
      <div className="col-md-2">
        <strong>Indv Price: <br></br>£{props.item.BskItemPrice}</strong> <br></br>
      </div>
      <div className="col-md-2">
        <strong>Total Price: <br></br>£{props.item.BskItemPriceTotal}</strong> <br></br>
      </div>
      <div className="col-md-3">
        <Link to={"/delete-item/" + props.item.id}>
          <button className="btn btn-danger" style={buttonStyle}>Delete Item</button>
        </Link>
      </div>
    </div>
  </div>
</div>

```

YOUR BASKET

	Product Name: Skint Knees Vinyl Record	Qty: 3	Indv Price: £59.99	Total Price: £179.97	Delete Item
---	---	-------------------------	-------------------------------------	---------------------------------------	---

Your Total Purchase is:

£179.97

[Checkout](#)

Database Setup – 10.2.2

Next was database set up for the mongoDB database. This was a lot simpler than the previous part of the application.

First of all I had to download the mongoDB from the MongoDB website. Once the database had downloaded to my desktop I had to make a few changes to the files before I could start using it. To start off I changed the name of the folder I downloaded to a more appropriate name that be used when in development. Once this was done I had to enter the folder and create a folder called Data. This is where all the database data will be stored. Once this has been done the next step was to create a folder inside this Data folder called db. This will be where all the databases will be saved. I needed to activate the MongoDB application by navigating to it in the terminal and activating the mongod.exe file in the folder 'bin' and point the application to where I will store the databases. For example, I had to enter 'C:\Users\steve\Desktop\MongoServer\bin\mongod.exe --dbpath=C:\Users\steve\Desktop\MongoServer\data\db'

```
2020-04-30T17:34:49.138+0100 I CONTROL [consoleTerminate] shutting down with code:12
PS D:\FYP\Application> C:\Users\steve\Desktop\MongoServer\bin\mongod.exe --dbpath=C:\Users\steve\Desktop\MongoServer\data\db
```

Once this is entered, I pressed the return key or 'enter' key and run the database. Now MongoDB was set up all I had to do was set up the database I would use. For this I would be using the MongoDB Compass.

Once installed the application was opened. Upon opening it would ask for the connection to the MongoDB server being used. In this box all I had to enter was 'mongodb://localhost:27017/' as this is the default port for MongoDB. Once this is done, press enter

Paste your connection string (SRV or Standard ⓘ)

mongodb://localhost:27017/

You have unsaved changes. [\[discard\]](#)

CONNECT

All that was left was to create the database. At the top of the screen there was a large green button labelled 'CREATE DATABASE'. I pressed this filled in the Input fields with 'FutureOfMusic' and pressed create database. At this point the database I would be using was finished and ready to use.

The screenshot shows the MongoDB Compass interface. On the left sidebar, it displays '8 DBS' and '15 COLLECTIONS'. The main area shows a table of databases. The 'FutureOfMusicDB' database is highlighted in blue. Below the table, there is a 'CREATE DATABASE' button.

Database Name	Storage Size	Collections	Indexes
FutureOfMusicDB	213.0KB	7	13
admin	20.5KB	0	1
config	24.6KB	0	2

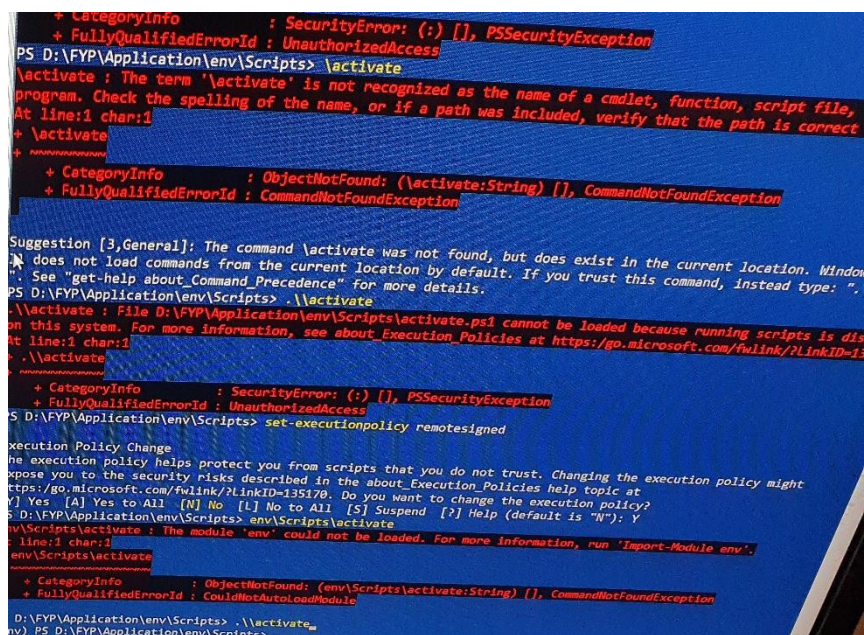
Back end Development– 12.2.3

Before the development of the back end could start there were a few things that had to be set up and installed first such as a few different package would need to be installed properly as well as A virtual environment would need to be set up ready to build the back end of the application in.

After installing Python onto my PC I needed to create the virtual environment to install all the package on in case of a problem with the installation as well as to protect the potential damage the primary installation. This is a copy of an existing version of Python with the chance to install other packages. This way during development it keeps the program safe from damage. If it wasn't done in an virtual environment and something went wrong it could have taken a while to fix before having to potentially start over.

To set up this virtual environment I had to open Windows PowerShell and start by checking the current installation of python with the 'pip -h' command. Once Python has been checked and is present I used the 'pip install virtualenv' command to install that package that allows me to create a virtual environment. Next is the creation of the virtual environment. First of all I had to create a folder in my D: drive in my FYP folder where I was going to save the back end and build the environment. This was completed using the command 'cd D:\FYP\Application'. To create a virtual environment, in the same PowerShell window, I entered the command 'virtualenv env'. The second part of this command is the name of your environment. It doesn't need to be env like mine. Once this command has finished, I needed to activate the application. This proved difficult as there were a few different sources online using different commands. Some required a command such as 'env\Scripts\activate' but more than often this would throw back an error.

This problem was soon solved by navigating into the environment using 'cd env\Scripts\' then using the command ./activate. On the first attempt this did fail but that was due to an admin error with the PC. When the command was activated Windows threw back an error stating that 'running scripts is disabled on this system. To fix this error I had to change the permission on my PC using the command 'set-executionpolicy remotesigned' then entering 'Y' when prompted. Once this was complete, I was able to use the activate command again to activate and enter the virtual environment to get the packages ready for development.



```
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

PS D:\FYP\Application\env\Scripts> \activate
\activate : The term '\activate' is not recognized as the name of a cmdlet, function, script file,
program. Check the spelling of the name, or if a path was included, verify that the path is correct
At line:1 char:1
+ ~~~~~
+ \activate
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (\activate:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CommandNotFoundException

Suggestion [3,General]: The command \activate was not found, but does exist in the current location. Windows
PowerShell does not load commands from the current location by default. If you trust this command, instead type: ".
\activate". See "get-help about_Command_Precedence" for more details.
PS D:\FYP\Application\env\Scripts> .\activate
.\activate : File D:\FYP\Application\env\Scripts\activate.ps1 cannot be loaded because running scripts is disabled
on this system. For more information, see about_Execution_Policies at https://go.microsoft.com/fwlink/?LinkID=135170
At line:1 char:1
+ ~~~~~
+ .\activate
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

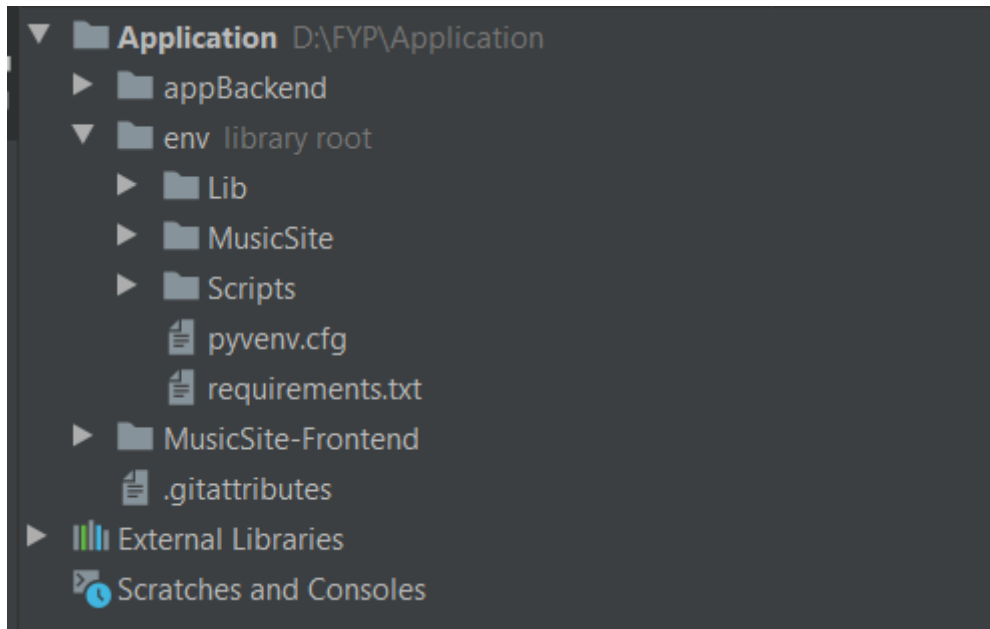
PS D:\FYP\Application\env\Scripts> set-executionpolicy remotesigned

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might
expose you to the security risks described in the about_Execution_Policies help topic at
https://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): Y
PS D:\FYP\Application\env\Scripts> env\Scripts\activate
env\Scripts\activate : The module 'env' could not be loaded. For more information, run 'Import-Module env'.
At line:1 char:1
+ ~~~~~
+ env\Scripts\activate
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (env\Scripts\activate:String) [], CommandNotFoundException
+ FullyQualifiedErrorId : CouldNotAutoLoadModule

D:\FYP\Application\env\Scripts> .\activate
env\ PS D:\FYP\Application\env\Scripts>
```

Now that the virtual environment is set up I was able to install all the packages needed to create the Django RESTful API. The first thing I needed to install was the Django framework. This was easily done by entering the command 'pip install django'. This will then check the latest version of python and install the Django framework. Next we also need to install the Django REST Framework. This is another Django framework that allows me to create a RESTful API. This is installed using a similar command to the last. The command is 'pip install djangorestframework'. Now that both Django frameworks are installed, I was now able to start with the development of the back end application.

To start off I first have to create a Django project inside the environment like how the React one was set up. Once this is done it will set up the file paths, file structure and complete a lot of basic functionality that



doesn't need to be changed. To do this I needed to enter another command into the PowerShell. The command is 'django-admin startproject MusicSite'. MusicSite at the end is the name of the project I am creating and for other users will not need to be this, but the rest of the command will have to be the same. This will then create a Django project folder inside the env ready to program. Once this was ready I was able to open the PyCharm application, like visual studio code but only for Python. Then using the file directory I navigated to the project I had created ready for development.

Now that I had an application to build off, I was getting closer to being able to develop the API. Before that could happen a few minor settings changes had to be made to make sure the application run correctly. First inside the MusicSite folder in settings.py I would need to include the rest framework, so that the application I build will be able to be a RESTful API.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
]
```


The next step I needed to complete was the connection the application to my MongoDB database. To do this first I needed to install the Djongo. This is a Django MongoDB connector to allow Django to connect to the database. The command for this is 'pip install djongo'. Once this has finished installing I needed to adjust the settings file again so that it used the same settings as the Mongo database does. Going back into the settings.py file I added the code on the right to the database settings so that they were able to connect.

```
DATABASES = {  
    'default': {  
        'ENGINE': 'djongo',  
        'NAME': 'FutureOfMusicDB',  
        'HOST': '127.0.0.1',  
        'PORT': 27017,  
    }  
}
```

Now that everything was set up I was now able to actually start building the functionality of the API. I started by creating an application for each of the tables I was going to use. This had to be done in the PowerCell Via commands again. First I had to navigate into the project folder using 'cd MusicSite'. Then using the command 'python manage.py startapp products' I was able to create a Django app for the products table. The last part of the command can be change depending on what you are using it for. Here I create one for products but using the same command I am able to create one for Basket and Purchases also. Each app in the Django will do something different and will connect to another part of the database. So, products will link with the products table, basket with the basket table and purchases with its table.

This will create app folders of those names inside the project folder MusicSite. These are basically classes and models as they will set up what data is being used, how it will be formatted as well as the functionality of the request like is it a GET, PUT, POST or DELETE and complete the corresponding programmed tasks.

Now that I've created all 3 apps for the API I had to include their config files into the installed apps so the applications knows to use them when the server is activated as well as what to do with them when they're requested. This is done like so:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'products.apps.ProductsConfig',  
    'basket.apps.BasketConfig',  
    'purchases.apps.PurchasesConfig',  
]
```

Now that these have been added to the `Installed_Apps` the application will be able to recognise them then they are needed. Next we need to set up CORS. CORS is a small bit of middleware used so that external applications, such as the front end, is able to send requests to the API. Without this it will not allow data to be sent back to the front end. To do this I first needed to install the CORS middleware package using the command `'pip install Django-cors-headers'`. Once this finished installing it had to be added to both the `Installed_Apps` in settings for the project as well as the `Middleware` part of the settings. Like so:

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'products.apps.ProductsConfig',  
    'basket.apps.BasketConfig',  
    'purchases.apps.PurchasesConfig',  
    'corsheaders',  
]
```

```
MIDDLEWARE = [  
    'django.middleware.security.SecurityMiddleware',  
    'django.contrib.sessions.middleware.SessionMiddleware',  
    'corsheaders.middleware.CorsMiddleware',  
    'django.middleware.common.CommonMiddleware',  
    'django.middleware.csrf.CsrfViewMiddleware',  
    'django.contrib.auth.middleware.AuthenticationMiddleware',  
    'django.contrib.messages.middleware.MessageMiddleware',  
    'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]
```

The middleware had to be placed further up the Middleware list so it was activated before the other middleware, especially before the common Middleware. Otherwise The cors headers wouldn't work and will still deny the front end the data it is requesting. Finally as part of the cors middleware I needed to set `CORS_ORIGIN_ALLOW_ALL` to `false` so that other external applications cannot access the API as well as adding the server port the front end is hosted on to the `CORS_ORIGINS_WHITELIST`. This will then give access to the API using its server ID and no other application.

```
CORS_ORIGIN_ALLOW_ALL = False  
CORS_ORIGINS_WHITELIST = (  
    'http://localhost:3000',  
)
```


Now all the security and packages needed are set up I was able to start building the models for the apps created earlier on. To start building the model of the products app I first had to navigate to the model.py file inside the products folder. This is where the model will be built. This will define what the object names are, what data type they will be, as well as any validation such as max length of the input, if it's allowed to be false, and what the default value of the input is.

```
from django.db import models
import datetime

class Product(models.Model):
    ProdName = models.CharField(max_length=100, blank=False, default='')
    ProdDesc = models.CharField(max_length=250, blank=False, default='')
    ProdStock = models.DecimalField(max_digits=2, decimal_places=0)
    ProdReleaseDate = models.DateField(default=datetime.datetime(1997, 9, 16))
    ProdPrice = models.DecimalField(max_digits=7, decimal_places=2)
    ProdFileName = models.CharField(max_length=70, blank=False, default='')
    ProdType = models.CharField(max_length=20, blank=False, default='')

```

The import of datetime works similarly to the JavaScript function Date(). This is a built-in function that allows the page it is used on to get the current date and input into an input area. It can also be used to enter a pre-set date as you can see above. This version of datetime sets the default date of ProdReleaseDate to 16/09/1997. The variable names have been linked to the model being developed. So, Prod would be product and the rest is a piece of product data such as name or Description. Notice as ID isn't in the model. This is because this framework adds the ID automatically when the data is added to the database. This was completed for apps Product, Purchases and Basket as all 3 will have their own set of objects to be entered to the database as well as different types of data.

Now that all 3 apps have a fully developed model, I needed to migrate them to the database so the tables are ready for when data is being sent to them. To do this I have to go back to the PowerCell and enter this command, 'python manage.py makemigrations products'. This needed to be done for all 3 apps to set up their models but instead of using products at the end it would be replaced with purchases and basket.

This creates a file called 0001_initial.py to show the migration of the model was a success. If changes to the model are needed to be made this command will have to be complete again which will create a new initial.py file with the new migration and models rules. If I now look inside this file I will see the object name, the validation around it as well as an automatically generated ID.

```

class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Product',
            fields=[
                ('id', models.AutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
                ('ProdName', models.CharField(default='', max_length=100)),
                ('ProdDesc', models.CharField(default='', max_length=250)),
                ('ProdStock', models.BooleanField(default=False)),
                ('ProdReleaseDate', models.DateField(default=datetime.datetime(1997, 9, 16, 0, 0))),
                ('ProdPrice', models.DecimalField(decimal_places=2, max_digits=7)),
                ('ProdFileName', models.CharField(default='', max_length=70)),
            ],
        ),
    ]

```

Once this is done, I then had to complete the command 'python manage.py migrate products'. This will then create the table in the database. Again this was done with all 3 applications but replacing the word products with the other apps such as purchases and basket.

basket_userbasket	1	195.0 E
django_migrations	11	95.1 B
products_product	14	305.4 E
purchases_purchase	29	315.7 E

Next is the serialization of the data. This needs to be added so the API knows how to serialize the data and deserialize the data from JSON. To do this first I had to create the a file called serializers.py in the apps folder. From there I needed to import the rest_framework as a serializer and import the model as itself. There were a few problems with this as I was trying to use products.models to import products. This was not needed and in fact caused errors. Because I was already in the right file directory the application was looking for a folder named products in products and then the file named models. Once I removed the 'products.' It worked fine as the model to serializer was already here. After this I had to build a serializer class named ProductsSerializer and create a meta for it including all the data that is stored in the model.

This is what will automatically populate the field in the model and set the validators when data is sent. The serializer looks like this

```
from rest_framework import serializers
from .models import Product

class ProductSerializer(serializers.ModelSerializer):
    class Meta:
        model = Product
        fields = (
            'id',
            'ProdName',
            'ProdDesc',
            'ProdStock',
            'ProdReleaseDate',
            'ProdPrice',
            'ProdFileName',
            'ProdType')

```

Again this was done with all the of the apps created earlier on but using the models they are part of as well as the specific model items specified in the models specific to them. They won't all look like the one above but will have the same build and features with the Meta subclass, the introduction of the model so they serializer knows what data and validation to use and the fields with all the specific fields create in the model and database. These names have to be EXACTLY the same. These files are case sensitive so using a capital in one and not another will cause errors later on when data is being sent or received from the API.

The next step would be creating the urls for each of these apps so the application knows what requests to expect and what to do with the application when it does receive this application. This is one of the simpler tasks. Once inside the app again I had to create the file urls.py to add these endpoints that will run functions later on when the URL is requested. This will import the Django.conf.urls as URI and the app

folder I'm in. Then I added the following URL to complete certain tasks later.

```
from django.conf.urls import url
from products import views

urlpatterns = [
    url(r'^api/products$', views.product_list),
    url(r'^api/products/(?P<pk>[0-9]+)$', views.product_detail),
    url(r'^api/products/instruments$', views.product_list_instruments),
    url(r'^api/products/equipment$', views.product_list_equipment),
    url(r'^api/products/records$', views.product_list_records)
```

Before I could carry on the project also has a file called `urls.py`. This is where I configured the application so it knew about the urls in products and the other apps so the requests could be completed

```
from django.conf.urls import url, include

urlpatterns = [
    url(r'^$', include('api.urls')),
    url(r'^$', include('products.urls')),
    url(r'^$', include('basket.urls')),
    url(r'^$', include('purchases.urls')),
```

Finally I would have to create the views for this applications. This was developed in the apps file, `views.py`. These are where all the functions are created that will run when one of the requests from the `urls.py` page are requested. Here I created a few nested if statements that would run depending on the request. Such as if there is no ID sent in the request it will complete the first nested if, if there is an ID it will complete the second tested and finally the last 3 functions would be completed if a specific product type is passed through to the request.

```

from django.shortcuts import render

from django.http.response import JsonResponse
from rest_framework.parsers import JSONParser
from rest_framework import status

from .models import Product
from .serializers import ProductSerializer
from rest_framework.decorators import api_view

@api_view(['GET', 'POST', 'DELETE'])
def product_list(request):
    if request.method == 'GET':
        products = Product.objects.all()

        ProdName = request.GET.get('ProdName', None)
        if ProdName is not None:
            products = products.filter(ProdName__icontains=ProdName)

        products_serializer = ProductSerializer(products, many=True)
        return JsonResponse(products_serializer.data, safe=False)
        # 'safe=False' for objects serialization
    elif request.method == 'POST':
        product_data = JSONParser().parse(request)
        products_serializer = ProductSerializer(data=product_data)
        if products_serializer.is_valid():
            products_serializer.save()
            return JsonResponse(products_serializer.data, status=status.HTTP_201_CREATED)
        return JsonResponse(products_serializer.errors, status=status.HTTP_400_BAD_REQUEST)

```

In the function where an ID is passed, this would be used to functions based on one particular product such as if I was deleting one, adding one or getting the information for one. The First function would be for collecting all information, deleting all information or adding a new item of data to the database. The last 3 are what it says on the tin. They get all of the items, with that specific product Type

```

@api_view(['GET', 'PUT', 'DELETE'])
def product_detail(request, pk):
    # find tutorial by pk (id)
    try:
        product = Product.objects.get(pk=pk)
        if request.method == 'GET':
            product_serializer = ProductSerializer(product)
            return JsonResponse(product_serializer.data)
        elif request.method == 'PUT':
            product_data = JSONParser().parse(request)
            product_serializer = ProductSerializer(product, data=product_data)
            if product_serializer.is_valid():
                product_serializer.save()
                return JsonResponse(product_serializer.data)
            return JsonResponse(product_serializer.errors, status=status.HTTP_400_BAD_REQUEST)
        elif request.method == 'DELETE':
            product.delete()
            return JsonResponse({'message': 'Product was deleted successfully!'}, status=status.HTTP_204_NO_CONTENT)
    except Product.DoesNotExist:
        return JsonResponse({'message': 'The Product does not exist'}, status=status.HTTP_404_NOT_FOUND)

@api_view(['GET'])
def product_list_equipment(request):
    product = Product.objects.filter(ProdType='equipment')

    if request.method == 'GET':
        product_serializer = ProductSerializer(product, many=True)
        return JsonResponse(product_serializer.data, safe=False)

```

Cannot Run Git

Once this is completed the with necessary functions and URLs for all 3 apps the djanog server can be turned on. Going back to the PowerCell I was now able to enter the command 'python manage.py runserver 8080'. This would then run the Django REST API. A few errors did show up on initial upload but these were due to an error with the model and serializer. The spelling and the capitalisation on some was off and caused a number of errors. Once checked the server was run again with no errors.

```

File "D:\FYP\Application\env\lib\site-packages\django\core\servers\basehttp.py", line 179,
    self.raw_requestline = self.rfile.readline(65537)
File "c:\users\steve\appdata\local\programs\python\python38-32\lib\socket.py", line 669, i
    return self._sock.recv_into(b)
ConnectionAbortedError: [WinError 10053] An established connection was aborted by the softwa
-----
[30/Apr/2020 16:52:08] "GET /api/basket HTTP/1.1" 200 183
[30/Apr/2020 17:12:31] "GET /api/products HTTP/1.1" 200 4214
[30/Apr/2020 17:12:32] "GET /api/products/1 HTTP/1.1" 200 263
[30/Apr/2020 17:20:21] "GET /api/basket HTTP/1.1" 200 183
(env) PS D:\FYP\Application\env\MusicSite> python manage.py runserver 8080

```

PROJECT SPECIFICATION - Project (Technical Computing) 2019/20

Student:	Steven Russell Tomlinson
Date:	16/10/2019
Supervisor:	Andrew Bissett
Degree Course:	Software Engineering BEng
Title of Project:	Evaluating a single page e-commerce site supported by an API compared a multi-page e-commerce

Elaboration

My project will be a single page music based e-commerce site that sells a range of different music-based products such as albums, singles, headphones, guitars etc. This page will allow users to create accounts so they can save items to their favourites as well as store more personal data about them. The front end will be developed using the React.js front end framework that allows for component reusability and allows for a quick and efficient web page experience. Doing this will eliminate the long URLs because building a single page application in React will allow for the page to dynamically change almost instantly instead of waiting for the server to load the next page. Every time a link is pressed a JS function will run and load the necessary components needed for that page as well as hiding all the components used in the previous page allowing for a smooth transition between pages with minimal loading time.

Also, not only that but this web application will be assisted using a Restful API developed in Django using a different language, Python. This API will allow different sources of information to pass through to it and output it to the front end, when requested, in JSON, allowing me to manipulate all the data used on this web application the same way. Also, unlike most multi page web applications all the data from the database is there all at once. The main use for the Restful API is that it acts like a search. When you load a specific page, it will only load specific data to do with the necessary components on that page instead of loading all the data and sorting through what you need and what you don't. Because it's assisting a JavaScript framework it will load new data from the database when the component is loaded without having to refresh the page.

At the end I will be evaluating the single page API assisted application against a multi-page application without an API to see what the benefits are of both of them as well as drawbacks and essentially work out which is better.

Project Aims

- Learn to develop a front end in React.js
- Learn to program Django using python
- Create a single page application using React.js
- Build a database for items, basket and purchases
- Build a Restful API using the Python framework, Django
- Pull data from a database and display using the restful API
- Post data to database using the rest API
- Complete this project using the most appropriate methodology.
- Evaluate the single page e-commerce application with API against a multi-page e-commerce application (e.g. Amazon)

Project deliverable(s)

The deliverables I will produced will include the following:

- A single page e-commerce web application built in React.js
- An API built with the python framework, Django
- Create 1 functional node.js server to host the front end on

- Create 1 functional back end server using Django to host the API

Action plan

Personal tasks	Deadline
Complete project specification	25/10/2019
Research the best methodology to use in order to complete my project	08/11/2019
Create and set-up GitHub version control for project	15/11/2019
Research the basics of the React.js framework	15/11/2019
Research how to connect the React page to a database	22/11/2019
Research how to display API data in from pre-existing APIs on the React page	29/11/2019
Document what I have learned about single page applications in React	06/12/2019
Research the basics of the programming language Python	13/12/2019
Research how to input and retrieve data from the database using Python	20/12/2019
Research how to develop an API using Python	27/12/2019
Document what I have found out about APIs in Python	03/01/2020
Start first major development of the project	17/01/2020
Finish React front end	31/01/2020
Finish building API	14/02/2020
Finish Connecting front end to the database using the API	21/02/2020
Ask a select few people to use the application and answer a questionnaire based on it	28/02/2020
Develop version 2 of the project based on the feedback from the questionnaires.	06/03/2020
Testing	13/03/2020
Document user testing. Add changes if needed	20/03/2020
Draft of Evaluation	27/03/2020

Module Tasks	Deadlines
Complete project specification	25/10/2019
Upload information review to BB	06/02/2020
Upload provisional contents page to BB	21/02/2020
Upload draft critical evaluation to BB	27/03/2020
Upload sections of draft report to BB	27/03/2020
Submit project body to BB	22/04/2020
Submit the Project Report (physical and electronic) and copies of the deliverable in the report and on BB or the Q drive	23/04/2020
Live demo of project	Before 12/05/2020

BCS Code of Conduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.

Signature: Steven Tomlinson

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.

Signature: Steven Tomlinson

GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire or participant testing materials. This form is available on the Bb site for the module.

Signature: Steven Tomlinson

Ethics

Complete the SHUREC 7 (research ethics checklist for students) form below. If you think that your project may include ethical issues that need resolving (working with vulnerable people, testing procedures etc.) then discuss this with your supervisor as soon as possible and comment further here.

Both you and your supervisor need to sign the completed SHUREC 7 form.

Please contact the project co-ordinator if further advice is needed.

RESEARCH ETHICS CHECKLIST FOR STUDENTS (SHUREC 7)

This form is designed to help students and their supervisors to complete an ethical scrutiny of proposed research. The SHU [Research Ethics Policy](#) should be consulted before completing the form.

Answering the questions below will help you decide whether your proposed research requires ethical review by a Designated Research Ethics Working Group.

The final responsibility for ensuring that ethical research practices are followed rests with the supervisor for student research.

Note that students and staff are responsible for making suitable arrangements for keeping data secure and, if relevant, for keeping the identity of participants anonymous. They are also responsible for following SHU guidelines about data encryption and research data management.

The form also enables the University and Faculty to keep a record confirming that research conducted has been subjected to ethical scrutiny.

For student projects, the form may be completed by the student and the supervisor and/or module leader (as applicable). In all cases, it should be counter-signed by the supervisor and/or module leader, and kept as a record showing that ethical scrutiny has occurred. Students should retain a copy for inclusion in their research projects, and staff should keep a copy in the student file.

Please note if it may be necessary to conduct a health and safety risk assessment for the proposed research. Further information can be obtained from the Faculty Safety Co-ordinator.

General Details

Name of student	Steven Russell Tomlinson
SHU email address	B6023855@my.shu.ac.uk
Course or qualification (student)	Software Engineering BEng
Name of supervisor	Andrew Bissett
email address	A.K.Bissett@shu.ac.uk
Title of proposed research	A single page music e-commerce site using a Restful API
Proposed start date	25/10/2019
Proposed end date	16/04/2020
Brief outline of research to include, rationale & aims (250-500 words).	My project will be a single page music based -commerce site that sells a range of different music-based products such as albums, singles. Headphones, guitars etc. This page will allow users to create accounts so they can save items to their favourites as well as store more personal data about them. The front end will be developed using the React.js front end framework that allows for component reusability and allows for a quick and efficient web page experience. Doing this will eliminate the long URLs because building a single page application in React will allow for the page to dynamically change almost instantly instead of waiting for

	<p>the server to load the next page. Every time a link is pressed a JS function will run and load the necessary components needed for that page as well as hiding all the components used in the previous page allowing for a smooth transition between pages with minimal loading time.</p> <p>Also, not only that but this web application will be assisted using a Restful API developed in Django using a different language, Python. This API will allow different sources of information to pass through to it and output it to the front end, when requested, in JSON, allowing me to manipulate all the data used on this web application the same way. Also, unlike most multi page web applications all the data from the database is there all at once. The main use for the Restful API is that it acts like a search. When you load a specific page, it will only load specific data to do with the necessary components on that page instead of loading all the data and sorting through what you need and what you don't. Because it's assisting a JavaScript framework it will load new data from the database when the component is loaded without having to refresh the page.</p> <p>At the end I will be evaluating the single page API assisted application against a multi-page application without an API to see what the benefits are of both as well as drawbacks and essentially work out which is better.</p> <ul style="list-style-type: none"> • Learn to develop in React.js • Learn to program Python • Create a single page application using React.js • Build a database for items, basket and purchases • Build a Restful API using Python • Pull data from a database and display using the restful API • Post data to database using the rest API • Complete this project using the most appropriate methodology. • Evaluate the single page e-commerce application with API against a multi-page e-commerce application (e.g. Amazon)
<p>Where data is collected from individuals, outline the nature of data, details of anonymisation, storage and disposal procedures if required (250-500 words).</p>	<p>As part of my research I will be asking individuals to use my website as a normal user and answer a small questionnaire on their experiences so I can refine the front web page as well as display information people would expect to see and hide what they wouldn't expect to see. On the questionnaire I will record the persons name, age and contact information in case I want to ask further questions. This will be stored on my local device <u>only</u>.</p> <p>Also, I will ask them to input user data on the page which record similar information including the D.O.B(date of birth) and the users name and address. Once the project is complete all the information from the participants in question will be deleted from the database and questionnaires destroyed to prevent anyone from using this information.</p> <p>Also as part of my research I will need to gather a range of album covers from bands to simulate the e-commerce side of the web page. This will not be used against the copy right of the album covers as album covers come under the fair use act. If they are used to help identify the band or album itself or sell the album for financial gain of the band the album covers can be used for this web application.</p> <p>These will be stored in a local file on my computer and will not be used to for anything more than to identify the band or promote sales of their album for their financial gain.</p>

I. Health Related Research Involving the NHS or Social Care / Community Care or the Criminal Justice Service or with research participants unable to provide informed consent

Question	Yes/No
<p>1. Does the research involve?</p> <ul style="list-style-type: none"> • Patients recruited because of their past or present use of the NHS or Social Care • Relatives/carers of patients recruited because of their past or present use of the NHS or Social Care • Access to data, organs or other bodily material of past or present NHS patients • Foetal material and IVF involving NHS patients • The recently dead in NHS premises • Prisoners or others within the criminal justice system recruited for health- related research* • Police, court officials, prisoners or others within the criminal justice system* • Participants who are unable to provide informed consent due to their incapacity even if the project is not health related 	No
<p>2. Is this a research project as opposed to service evaluation or audit?</p> <p><i>For NHS definitions please see the following website</i> http://www.hra.nhs.uk/documents/2013/09/defining-research.pdf</p>	No

If you have answered **YES** to questions **1 & 2** then you **must** seek the appropriate external approvals from the NHS, Social Care or the National Offender Management Service (NOMS) under their independent Research Governance schemes. Further information is provided below.

NHS <https://www.myresearchproject.org.uk/Signin.aspx>

* All prison projects also need National Offender Management Service (NOMS) Approval and Governor's Approval and may need Ministry of Justice approval. Further guidance at:

<http://www.hra.nhs.uk/research-community/applying-for-approvals/national-offender-management-service-noms/>

NB FRECs provide Independent Scientific Review for NHS or SC research and initial scrutiny for ethics applications as required for university sponsorship of the research. Applicants can use the NHS pro-forma and submit this initially to their FREC.

2. Research with Human Participants

Question	Yes/No
Does the research involve human participants? This includes surveys, questionnaires, observing behaviour etc.	Yes
Question	Yes/No
<p>1. Note If YES, then please answer questions 2 to 10</p> <p>If NO, please go to Section 3</p>	-

2. Will any of the participants be vulnerable? <i>Note: Vulnerable' people include children and young people, people with learning disabilities, people who may be limited by age or sickness, etc. See definition on website</i>	No
3. Are drugs, placebos or other substances (e.g. food substances, vitamins) to be administered to the study participants or will the study involve invasive, intrusive or potentially harmful procedures of any kind?	No
4. Will tissue samples (including blood) be obtained from participants?	No
5. Is pain or more than mild discomfort likely to result from the study?	No
6. Will the study involve prolonged or repetitive testing?	No
7. Is there any reasonable and foreseeable risk of physical or emotional harm to any of the participants? <i>Note: Harm may be caused by distressing or intrusive interview questions, uncomfortable procedures involving the participant, invasion of privacy, topics relating to highly personal information, topics relating to illegal activity, etc.</i>	No
8. Will anyone be taking part without giving their informed consent?	No
9. Is it covert research? <i>Note: 'Covert research' refers to research that is conducted without the knowledge of participants.</i>	No
10. Will the research output allow identification of any individual who has not given their express consent to be identified?	No

If you answered **YES only** to question **1**, the checklist should be saved and any course procedures for submission followed. If you have answered **YES** to any of the other questions you are **required** to submit a SHUREC8A (or 8B) to the FREC. If you answered **YES** to question **8** and participants cannot provide informed consent due to their incapacity you must obtain the appropriate approvals from the NHS research governance system. Your supervisor will advise.

3. Research in Organisations

Question	Yes/No
1. Will the research involve working with/within an organisation (e.g. school, business, charity, museum, government department, international agency, etc.)?	No
2. If you answered YES to question 1, do you have granted access to conduct the research? <i>If YES, students please show evidence to your supervisor. PI should retain safely.</i>	N/A
3. If you answered NO to question 2, is it because: A. you have not yet asked B. you have asked and not yet received an answer C. you have asked and been refused access. <i>Note: You will only be able to start the research when you have been granted access.</i>	N/A

4. Research with Products and Artefacts

Question	Yes/No
1. Will the research involve working with copyrighted documents, films, broadcasts, photographs, artworks, designs, products, programmes, databases, networks, processes, existing datasets or secure data?	Yes
2. If you answered YES to question 1, are the materials you intend to use in the public domain? Notes: 'In the public domain' does not mean the same thing as 'publicly accessible'. <ul style="list-style-type: none"> Information which is 'in the public domain' is no longer protected by copyright (i.e. copyright has either expired or been waived) and can be used without permission. Information which is 'publicly accessible' (e.g. TV broadcasts, websites, artworks, newspapers) is available for anyone to consult/view. It is still protected by copyright even if there is no copyright notice. In UK law, copyright protection is automatic and does not require a copyright statement, although it is always good practice to provide one. It is necessary to check the terms and conditions of use to find out exactly how the material may be reused etc. <p>If you answered YES to question 1, be aware that you may need to consider other ethics codes. For example, when conducting Internet research, consult the code of the Association of Internet Researchers; for educational research, consult the Code of Ethics of the British Educational Research Association.</p>	No
3. If you answered NO to question 2, do you have explicit permission to use these materials as data? If YES, please show evidence to your supervisor.	Yes
4. If you answered NO to question 3, is it because: A. you have not yet asked permission B. you have asked and not yet received an answer C. you have asked and been refused access.	A/B/C

Adherence to SHU policy and procedures

Personal statement	
I can confirm that:	
– I have read the Sheffield Hallam University Research Ethics Policy and Procedures	
– I agree to abide by its principles	
Student	
Name: Steven Russell Tomlinson	Date: 20/10/2019
Signature:	
Supervisor or other person giving ethical sign-off	
I can confirm that completion of this form has not identified the need for ethical approval by the FREC or an NHS, Social Care or other external REC. The research will not commence until any approvals required under Sections 3 & 4 have been received.	
Name: Andrew Bissett	Date: 21/10/2019
Signature:	