# UYR Events

Contents

[UYR Events 1](#_Toc179801563)

[Aim of the Project 1](#_Toc179801564)

[Analysis and Design 1](#_Toc179801565)

[Implementation 4](#_Toc179801566)

[Using Test Data 6](#_Toc179801567)

[Displaying Data 7](#_Toc179801568)

[Products Page and Basket 11](#_Toc179801569)

# Aim of the Project

The aim of this project is to create an events page for my company. This task was given to me by my Technical Director and Managing Director, with the hopes of expanding into a different market and increasing sales. This will be a full stack application that will allow users to create and organise their events. Along with this, we will display our products that can then be purchased for the events. The website will require login information and a basket for checking out. For this project, I will be using a React frontend, including JavaScript and HTML, and a C# backend API. I will also be using the Bootstrap framework for styling purposes.

# Analysis and Design

First, I created wireframes for my design. The designs had to be clear and intuitive to enhance the user experience. Below are some examples of the wireframes I created:

A screenshot of a computer

Description automatically generated

Figure : Events Page

This will be the page that displays all the events a user has. Users will quickly be able to see information as each event card, seen in the middle, will show the event title, hosts, date, time, location, how many are invited and how many have accepted invitation. At the bottom of the card will be the buttons to edit the information of an event or to delete the event. Above the events is button to add a new event, which will trigger a modal to fill in the new information. To the left of the page will be the filters, which will help users find information quickly, leading to a positive user experience. To the right will be suggested products that when clicked will take you to that product page. This will hopefully improve conversion rates.

A screenshot of a computer

Description automatically generated

Figure : Product Page

Above is the product page where a user will be able to choose the type, size and quantity of a product, along with their associated prices. On the left, users will be able to review a preview of the product before pressing the ‘Add to Basket’ button.

The header of all my pages will be a navbar, where users can swiftly navigate through each section of the website. Within the navbar will be buttons to navigate to the basket and the login page for a user. Also, for pages that it applies to, there will be breadcrumbs that indicate the page’s location within its navigational hierarchy, resulting in a greater user experience and adhering to the WCAG guideline of ‘Help users navigate and find content’. Here I include a footer section which may show support information such as contact numbers and email addresses.

A screenshot of a computer

Description automatically generated

Figure 3: Basket

Lastly, here I am showing the users basket with all the products they have added and their associated costs. At the bottom will be the subtotal, delivery costs and final total of all goods in the basket. On the right-hand side is where the user will add their address and finally checkout to pay. The basket will need global state as users may be adding products to it from multiple different pages.

I plan to do this project with the support of my line manager and fellow colleagues who have more experience in aspects such as databases and C# backend API’s.

# Implementation

The first thing I implemented into my application was the routing paths that I will require. This gave me a basic idea of how users would navigate through my multi page app. I did this by importing in the JavaScript framework, React Router.

A screen shot of a computer program

Description automatically generated

Figure : Use of React Router

In figure 4 you can see how I have separated out each individual page, resulting in cleaner, more organised code. I have also determined the URL of each page with a relevantly labelled path.

On lines 23 and 34, you can see I have created a Header and Footer component. These have been purposely placed there, so that they are persistent throughout my website.

A screenshot of a computer

Description automatically generated

Figure 5: Header & Footer Design

In figure 5 above, you can see the design I have for the Header and Footer components. The layout of the elements has taken inspiration from one of our existing sites for brand consistency.

## Using Test Data

In order to progress and test my application I realised I would need some test data. To do this I decided on using JSON Server, which would allow me to create a mock REST API for my application.

A screen shot of a computer

Description automatically generated

Figure 6: JSON Server Data

In figure 6 you can see the data that I have inputted to create some events. I have also created properties for messages, users, products and a basket, for me to use as endpoints.

The live data can be seen below in figure 7, whereby I am previewing the data on the events endpoint.

A screenshot of a computer program

Description automatically generated

Figure 7: JSON Data from Events Endpoint

## Displaying Data

My next task was trying to display the data on my webpage. I decided on using Axios, a promised-based HTTP Client, to make my API calls.

I made a utils.js folder to separate out my API calls, allowing for reusability, and added my GET request. Seen below (figure 8). This makes a call to my events endpoint and receives a promise which is then returned.

A screen shot of a computer

Description automatically generated

Figure 8: Axios GET Request

In order to display the returned data, I used the React Hook, useEffect (see figure 9). This runs on first render and triggers my API call. Once this runs, I used the React Hook, useState, to set my events data to the returned value.

A screen shot of a computer program

Description automatically generated

Figure 9: useEffect and useState Hook

Finally, in the body of my Events component, I can now map over the data and provide it as props to my EventCard component for formatting. The result can be seen below in figure 10.

A screenshot of a computer

Description automatically generated

Figure : Displaying my Events Data

Using Bootstrap, I have styled each event within a card, applying suitable padding and margin between each element. I have also created a button group which has the edit and delete buttons. The layout has been set using Bootstraps grid system, which is built with flexbox. This means that the elements on the page change dynamically to smaller or larger screen sizes and use a mobile-first approach.

At the top of this page, I have added a ‘Add New Event’ button. When pressed, this produces a modal that allows a user to input new event data. The modal can be seen below in figure 11.

A screenshot of a computer

Description automatically generated

Figure 11: New Event Modal

Once submit is pressed it triggers an addNew function which takes the form data and then uses two Axios POST requests to submit the new event and any message data. Below in figure 12 and 13, you can see the code for the addNew function and the POST requests.

A screen shot of a computer program

Description automatically generated

Figure 12: addNew Function

A screen shot of a computer program

Description automatically generated

Figure 13: New Event POST Requests

When sending POST requests, the data is automatically set an ‘ID’ property so each individual bit of data can be identified. This ID is useful for looping through data in React, as each child in a list should have a unique ‘key’ prop as a way for React to handle DOM changes. I have added an ‘eventID’ property, so that I can make requests to the message’s endpoint by eventID. I have also added the ‘sender’ property and a ‘date-time’ property so messages can be sorted. Below you can see the result of my new event (figure 14).

A screenshot of a computer

Description automatically generated

Figure 14: New Event

## Products Page and Basket

The other aspect of my application is the sale of our products. To implement this, I added the information for the products that we sell, into my JSON data. I then made a GET request to my products endpoint and iterated over them in React, creating Product Cards in the process. I downloaded images from the internet just as templates and the result can be seen below (figure 15).

A screenshot of a social media post

Description automatically generated

Figure : Products Page

Here you can see the products page (figure 15). Each product has a title and starting price, as well as an ‘Info’ button, which will take you to the specific product page to decide quantity and image. For accessibility and adherence to WCAG guidelines, I have added alternative text to each image displayed so that screen readers can provide information about the images purpose to the reader. To the left of this page, you can see the filters which will help the user narrow down the products they require, resulting in a greater user experience (see figure 16). This filters the array of products and then sets the product data to the result.

A screenshot of a social media post

Description automatically generated

Figure 16: Filtered Products by Correx

After clicking on a product, it will take you to the product info page. This will allow a user to select the quantity they would like. Changing the quantity will give the user a preview of the price, and at that point, they can select ‘Add to Basket’. Adding to basket is a POST request to the user’s, basket endpoint. After adding a product, the basket icon will display a badge indicating how many products they have.

A screenshot of a computer

Description automatically generated

Figure 17: Product Info Page

The basket is where the user will be able to see all the products they have added, as well as the quantities they have chosen. I created the basket using the React Hook, useContext in the App.js of my application. This in effect, creates global state, so all components can access it. This was important to avoid prop drilling, which is viewed as bad practice, because it creates redundant code and impacts readability.

Within the basket, users will be able to edit the quantity of their items or delete the product entirely. The Axios requests for these can be seen below in my utils file (figure 18).A computer screen shot of code

Description automatically generated

Figure 18: Axios Requests for the Basket

A screenshot of a computer

Description automatically generated

Figure : Basket

A preview of the basket can be seen above (figure 19). On the right is where a user can change their delivery address and checkout. Each item in the basket has buttons to change the quantity, which is then reflected live in the areas where cost is shown. I have also added some basic validation for when a user has reached the spending level to qualify for free delivery. When the basket subtotal is below £50, in this case the validation looks like figure 20. Hopefully, a feature like this would encourage a user to spend more, resulting in higher AOV figures.

A screenshot of a computer

Description automatically generated

Figure : Basket Validation

# Testing and Error Handling

When navigating through my site I found I was experiencing errors which would force my application to timeout. Through manual testing, I realised this was due to the asynchronous API calls I was making, which was then returning null when trying to reference them in the body of my code. An example of this can be seen below in figure 21.

A screenshot of a computer

Description automatically generated

Figure 21: Runtime Error I was experiencing

To combat this, I added error and loading handling to all my pages where there may be asynchronous operations happening. I later realised I would need this throughout my whole application as network errors could also occur. My Error and Loading components can be seen below (figure 22 & 23).

A screen shot of a computer error

Description automatically generated

Figure 22: Error Component

A screen shot of a computer program

Description automatically generated

Figure 23: Loading Component

A screen shot of a computer screen

Description automatically generatedHere you can see an example of my product page with the Error and Loading handling being conditionally applied (figure 24). If an API call were to throw an error, it would be caught, and my Error component would be rendered. This displays a message to the user indicating the error code and the type of error.

Figure 24: Product Page with Error and Loading Handling

The Loading component will get shown whilst the API call is being made. For example, if a user were suffering from poor connectivity issues this would get displayed whilst they are waiting. Once both conditions have been met; there are no errors and the request has finished loading, the body of my code will be rendered. Hopefully, this type of error handling will avoid user frustration as they are given feedback on the issue to avoid uncertainty. Also, for accessibility needs, it enables screen readers to read out the clear and concise messages and encourage retention to the site.

# Revisions

In order to maintain my code and track the changes, I used the version control platform, GitHub. Using GitHub branching system, I was able to test different approaches and designs to my codebase. Once happy with the changes, I was able to merge and push these changes to the cloud. All of these changes were documented with commit messages that serve as a log for me, or others in the future to look at and understand.

# Security

# Conclusion