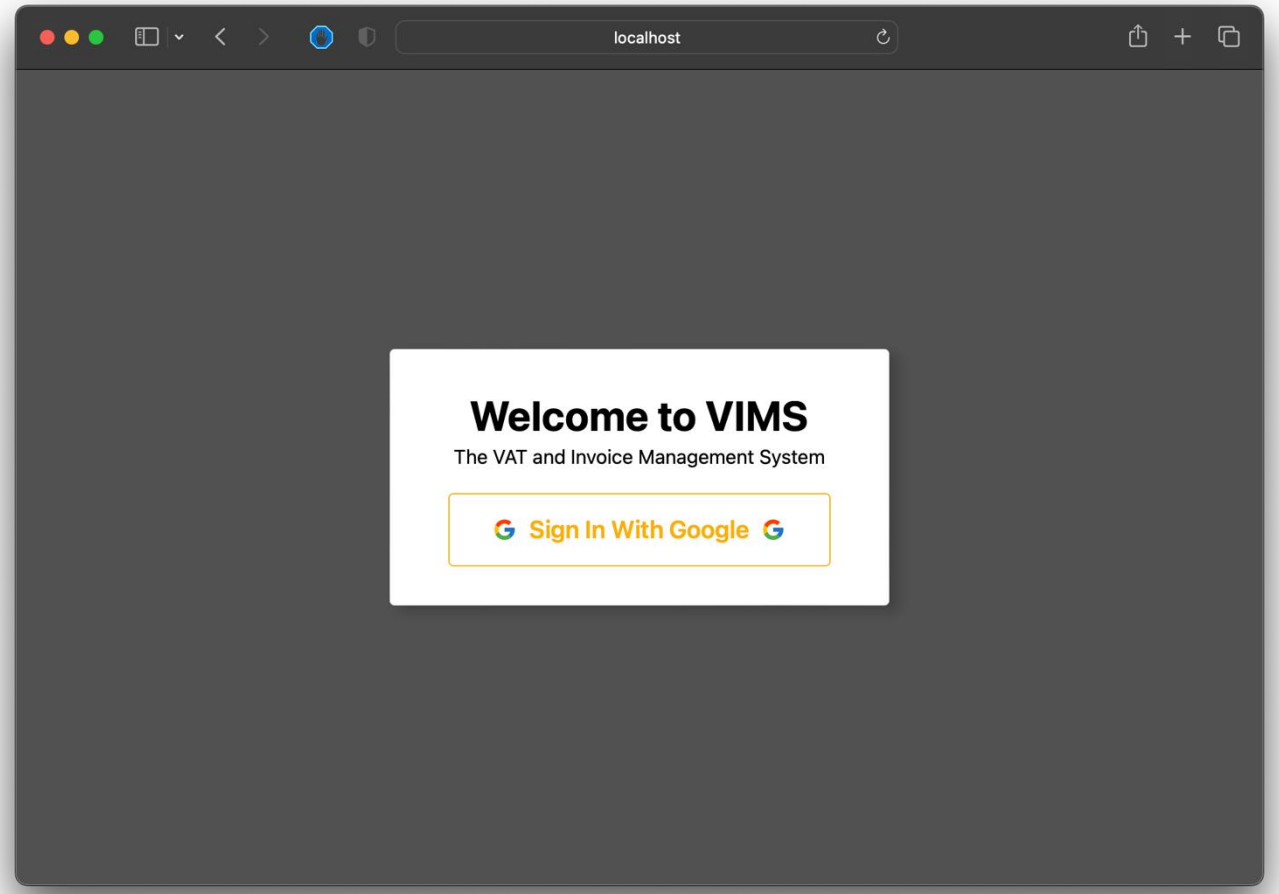
Daniel Jacob Spiers

Computer Science A Level Project

VIMS (VAT & Invoice Management System)

New College Pontefract

Centre No. 38185



Analysis of the Problem

Problem Identification

My stakeholders need a piece of software that can manage the invoices and submit quarterly VAT returns to HMRC for their bus and coach glazing business. This cannot be done manually because of the Making Tax Digital (MTD) scheme which requires VAT returns to be submitted online with registered accounting software. My program will be registered with this scheme and will interact with HMRC's API services to submit return through there.

My stakeholders also need a centralised invoice management system. This can be done through software such as Excel but with limited capabilities. For example, Excel can be used to keep a record of profits and loss, but data is displayed in a hard to read format. My program will format invoices into a more readable state and allow the user to search for specific ones using date ranges.

Justification for Computational Approach

Although invoice management can be done without a computer, there would be vast amounts of paper that needs to be stored in an organised filing cabinet which takes up space and is time consuming to sort through. By organising invoices digitally in a database, sorting/filtering through all past invoices can be done instantly with minimal effort put in by the user.

Also, through hosting the application on the web, invoices can be accessed from anywhere with an internet connection on any device. Data security will also be higher as the database will be hosted in the cloud. Encryption is not feasible with paper documents but on a computer, multiple layers of security can be added without adding much - if any - cost.

Another reason for this is that VAT returns can now only be submitted digitally with HMRC registered accounting software, most of which are paid. There are some free ones, but they use adverts to generate revenue which crowd the screen - or have outdated UI - reducing user experience. By commissioning a piece of software from me, my stakeholders are giving a one-time payment rather than a monthly subscription.

Stakeholders

I am developing my solution for my parent's bus and coach glazing business: Elite Specialist Glazing. The current software they use is QuickBooks which they often complain is too complicated for what they need.

Primary Stakeholder

My primary stakeholder is my mum, Karen Plant. She handles the financial side of my parent's business and interacts with accounting software and invoices daily. Therefore, formatting of invoices is of prime importance. Current solutions do not have the required fields that she needs for how she lays out invoices. There are also superfluous fields that do not relate to her on the forms. Through a bespoke solution, the fields can be tailored to exactly how she needs them. She will view invoices daily and submit a VAT return every 3 months. All of this will be done from her office. QuickBooks requires a £24 monthly subscription which is an unnecessary cost.

Secondary Stakeholder

My secondary stakeholder is my stepdad, Steve Plant. He is the fitter for their company which means he is the one who interacts with customers and fits the windows for them. This by extent means he fills out all the invoices. This tends to be upwards of 3 a day so he needs filling out the forms to be efficient. He travels all over most of the country and goes as far as Northampton or South Wales to do jobs, so portability is a major factor for him.

Research

Choosing a backend technology

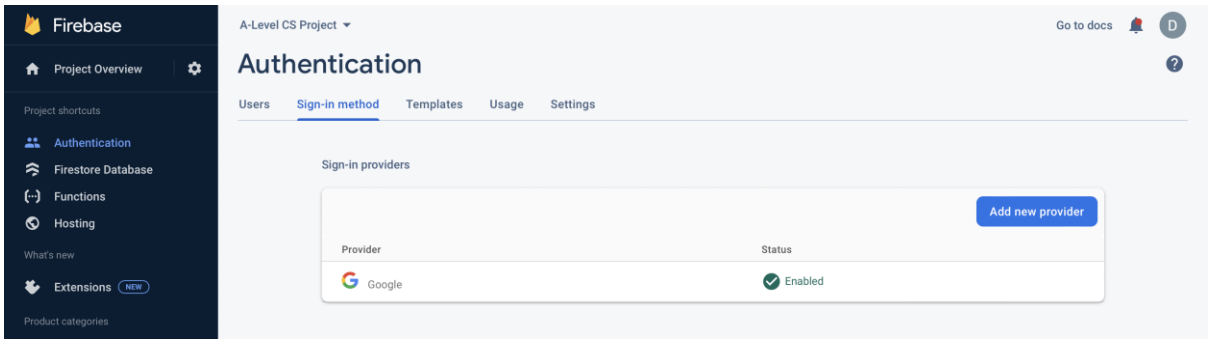
The first part of research I did was my choice of framework for the app's backend. Initially, I planned to write the backend in python due to its ease of use and my familiarity with the language. Going down this route, my choice of frameworks were Django and Flask.

Django is a full-stack web framework which has built in features that include: an admin framework to enable easy management and enforcing of permissions & access; the ability to divide a project into multiple small applications which allows for easy separation of the different sections of the program, which increases code readability and makes debugging easier as each app can be tested separately to isolate problems.

I chose Django over Flask as, even though Flask is lighter weight code wise, database and admin support is limited.

However – As time went on – I have changed my backend decision to Google's firebase (written in JavaScript) as they supply an easy-to-use database SDK and options for hosting the web app. Another thing that I like about firebase is their authentication, I can give my stakeholders the option to sign in using their google account which removes the need for them to remember a separate username and password for logging in.

Firebase UI and documentation on auth providers.



Sign in existing users

Create a form that allows existing users to sign in using their email address and password. When a user completes the form, call the `signInWithEmailAndPassword` method:

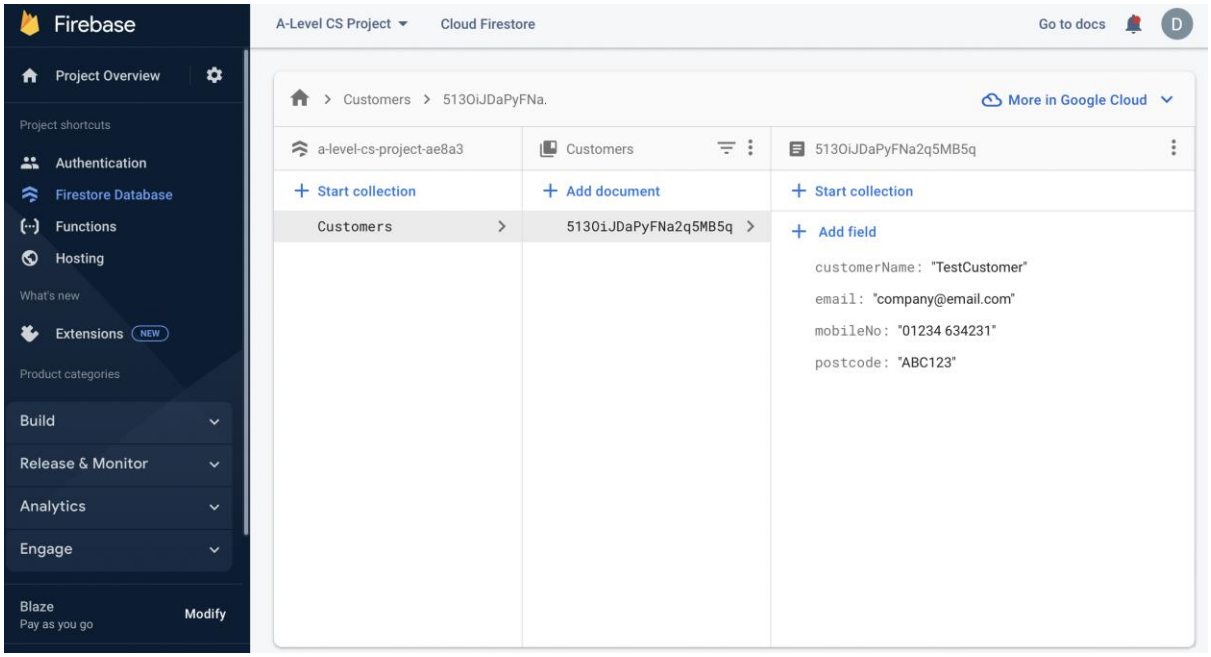
Web version 9 (modular) | Web version 8 (namespaced)

Learn more about the tree-shakeable Web v9 modular SDK and upgrade from version 8.

```
import { getAuth, signInWithEmailAndPassword } from "firebase/auth";

const auth = getAuth();
signInWithEmailAndPassword(auth, email, password)
  .then((userCredential) => {
    // Signed in
    const user = userCredential.user;
    // ...
  })
  .catch((error) => {
    const errorCode = error.code;
    const errorMessage = error.message;
  });
```

auth_signin_password.js

Firebase's database web service – Cloud Firestore – uses a NoSQL database which means that data is stored in collections and documents rather than tables and records.



Another key factor of choosing a backend was finding a place to host the site. Django doesn't offer anything out of the box, and you need to set it up with a third party hosting company. Firebase, however, allows you to deploy the site on Google's servers using one command. Pricing was a major factor for hosting the site as well. Firebase offers two plans: Spark (free) and Blaze (pay as you go).

Firebase plans pricing comparison

Products	No-cost Spark Plan Generous limits to get started	Pay as you go Blaze Plan Calculate pricing for apps at scale ✓ No-cost usage from Spark plan included*
A/B Testing	No-cost	
Analytics	No-cost	
App Distribution	No-cost	
App Indexing	No-cost	
Authentication		
Phone Auth - US, Canada, and India	10k/month	$0.01/verification
Phone Auth - All other countries	10k/month	$0.06/verification
Other Authentication services	✓	✓
With Identity Platform		
Monthly active users	50k/month	No-cost up to 50k MAUs Then Google Cloud pricing
Monthly active users - SAML/OIDC	50/month	No-cost up to 50 MAUs Then Google Cloud pricing

Cloud Firestore		
Stored data	1 GiB total	No-cost up to 1 GiB total Then $0.108 per additional GiB
Network egress	10 GiB/month	No-cost up to 10 GiB/month Then Google Cloud pricing
Document writes	20K writes/day	No-cost up to 20K writes/day Then Google Cloud pricing
Document reads	50K reads/day	No-cost up to 50K reads/day Then Google Cloud pricing
Document deletes	20K deletes/day	No-cost up to 20K deletes/day Then Google Cloud pricing
Cloud Functions		
Invocations	*Not applicable*	No-cost up to 2M/month Then $0.40/million
GB-seconds		No-cost up to 400K/month Then Google Cloud pricing
CPU-seconds		No-cost up to 200K/month Then Google Cloud pricing
Outbound networking		No-cost up to 5GB/month Then $0.12/GB
Cloud Build minutes		No-cost up to 120min/day Then $0.003/min
Container storage		No-cost up to 500MB of storage Then $0.10/GB/month
		*Pricing varies based on location
Hosting		
Storage	10 GB	$0.026/GB
Data transfer	360 MB/day	$0.15/GB
Custom domain & SSL	✓	✓
Multiple sites per project	✓	✓

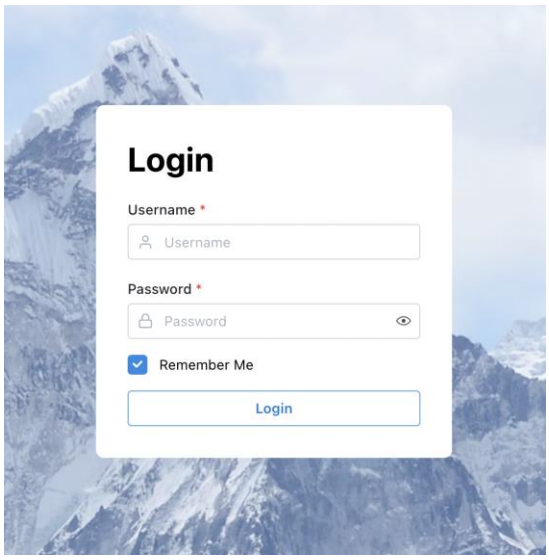
I will need Cloud Functions to execute the authentication commands for HMRC, so I decided to use the blaze plan.

Choosing a frontend technology

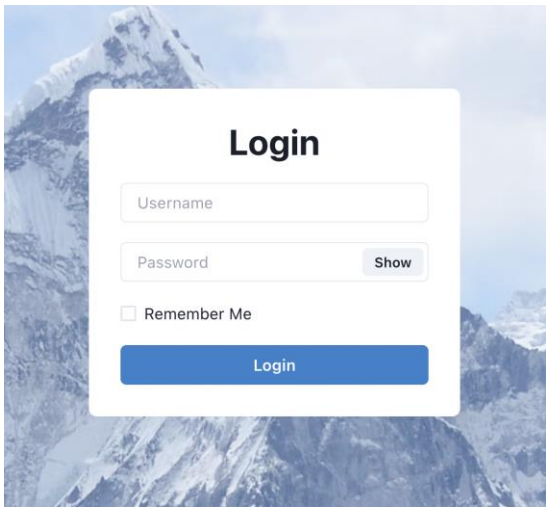
For the app's frontend, I chose to use react.js as it is simple to learn and is one of the most popular / most used frontend frameworks in the industry. It uses a modular approach to frontend design through components, which are sub-structures in the main application tree. In theory, the whole frontend could be written in plain HTML, CSS, and JS but this would be extremely time-consuming and complex to pull off. React uses a syntax that is a superscript of JavaScript – JSX – which allows for HTML to be written directly in the file, allowing me to write HTML with the full power of JavaScript readily accessible.

I will also use a UI library for the frontend to save me time in trying to make the app look good. I was indecisive between 2 libraries for the UI, but I only want to use one to make the app design look consistent. These libraries were Mantine or and Chakra UI. Both libraries have a unique style and provide standard UI components such as text input fields, buttons, dropdown menus, notifications, etc. The final decision came down to which style my stakeholders prefer more. To decide this, I made a mock-up login form using each library as shown below:

Mantine



Chakra UI

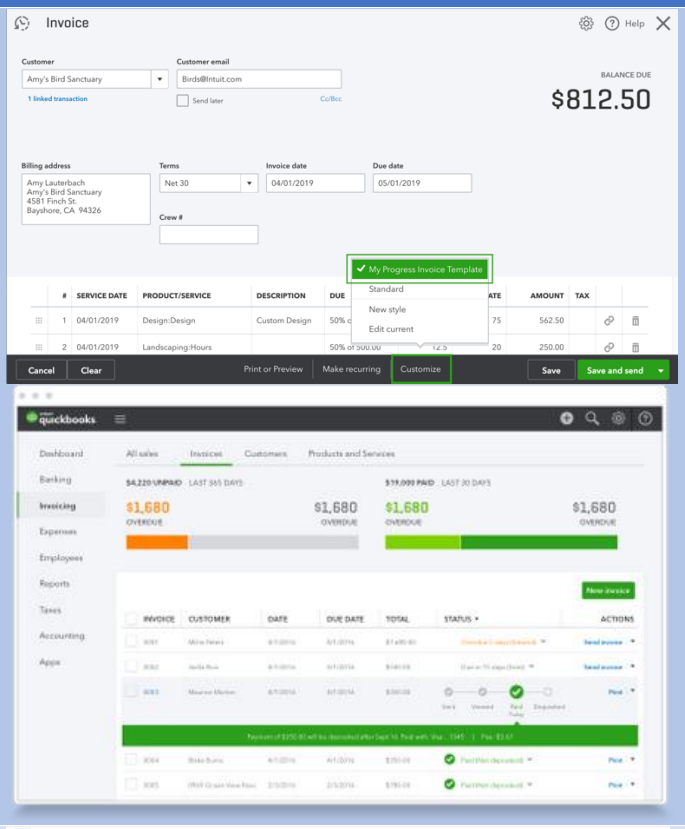
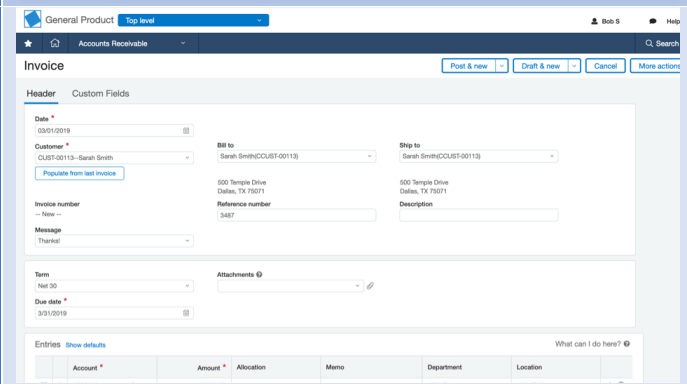


The chosen UI library was Mantine, so that is the one that will be used throughout development.

HMRC API usage

To submit VAT returns, my app will need to interface with HMRC's VAT API. Their documentation on their app shows all the API's endpoints and what they are used for. For me, I will only need to access the "Submit a VAT return" endpoint. I will need to follow their authentication protocols to access this endpoint which requires a token with the "write:vat" scope. I will then need to submit a post request to this endpoint with JSON data in the body in the format that is required. The body of the http request I need to send to their API mirrors a standard 9 box vat return.

Existing Solutions

Solution Name	Screenshots	Useful Features
Intuit QuickBooks	 	- Table showing items/services - The ability to search through invoices and see how much is owed
Sage		- Customer selection field (auto fills in details) - Cloud integration allowing access from anywhere

Specify the Proposed Solution

Essential Features

The main feature of the solution is integration with HMRC's VAT API. There is a workflow that will be followed each time the user submits a VAT return. The user will enter the required details into the form and click submit. Then, the system will check that all data in the form is valid (no letters where numbers should be, etc.). The user will then be redirected to HMRC's website where they will log in to their account with them. Once login is successful and the user has approved my software for submitting their VAT returns, they will be redirected back to my website with an access code. My website will then send a request back to HMRC with this access code to get a token. This token is stored in my database and can be used for all further requests to their API for the next 3 hours. My app will package the data from the user form and the access token received, to submit a request to HMRC's submit a VAT return endpoint.

The invoice management section of the website will allow the user to create, update, and edit invoices. Deletion of invoices will not be allowed so an accurate cash flow can be recorded. The user will also be able to create, update, edit, and delete customers to use for the invoices.

When the user creates a customer, they will fill out the details into a form such as their name, address, contact number and email. The system will then check that the email is in the correct form and the mobile number only contains numbers. They will be prompted with a notification when they press the submit button and another when the customer is added to the database and the page is refreshed.

When the user creates an invoice, they will fill out the details specific to the invoice and select a customer that the invoice is for. The system will then automatically fill in the email field. When adding an item to the invoice, another form will be presented that the user adds the service name, date, quantity, and description to along with price and VAT rate. The form will display a subtotal for this item. The invoice will display a net, vat, and total price along with an option to set the invoice as paid. When the user presses submit, the system checks that the inputted email is of the correct form and that at least one invoice item is present, then sends the invoice to the database and refreshes the page.

When editing a customer, the user will select their desired customer from a table of customers in the database. When the update page loads, the system will get the selected customer's data from the database and populate the fields in the form. The user will then be able to edit the customer's data and send the updated customer back to the database to be stored. The same goes for updating invoices but invoice items and the invoice number will not be able to be changed as this defeats the point of invoices by disrupting a linear flow of cash.

When deleting a customer, the user will be prompted to check that they meant to click the delete button and prevent accidental deletion of customers. A deletion request will then be sent to the database and the customer record deleted.

Database wise there will need to be three tables:

- A table containing all customers and their details. This will be used by my stakeholders to fill out invoice details more efficiently and allow them to look up a customer's details if they need to contact them.

- A table containing all invoices and their details. This will allow my stakeholders to keep track of how much money is owed and from whom. Also, all invoices are permanently stored so a full cash flow can be visible in the case of an audit.

- A table containing each user's access token, when it was created, and how long it is valid for.

Limitations

- As the application is hosted on the internet, the user cannot access the program if they don't have an internet connection. My secondary stakeholder travels all over the country, sometimes to places with no signal. So, he will have to fill out the invoice when he reconnects to the internet. This isn't much of a concern as he can just write down the costs on a piece of paper.

- Invoices cannot be formatted into PDF files and emailed to customers as in my research, I could not find a module that performs this task how my stakeholders want. This means I would have to write my own which would take too much time to implement. Therefore, my stakeholders will have to manually format invoices and email them to customers, which can be time consuming. Initially, this would limit them massively. But if they get a template to fill in, it won't be much of an issue.

- Another limitation is that VAT rates will not update automatically if there is an increase. The current VAT rate is 20% which could increase. This means that my stakeholders will have to go into the code and change the VAT rate variable if/when there is a change. This could potentially be a large limitation if they forget to change it. But if they remember, everything will be fine.

Software and Hardware Requirements

SOFTWARE/HARDWARE USED	FUNCTION AND JUSTIFICATION OF USE
Firebase	This is the SDK I will use to develop, host the database, and host the web app itself.
Cloud Firestore	This is a web service provided by Firebase, It allows for the creation of a NoSQL database which can be accessed in real time, keeping the currently loaded page up to date.
Cloud functions	This is another web service provided by Firebase which lets you write your own custom functions that can be stored securely in the cloud. I will use this for authenticating the user with HMRC as doing this on the frontend would be incredibly insecure.

Visual Studio Code (IDE)	I will need an IDE to develop the app. I picked VScode because it is the main IDE I use. It also has extensions for React.js syntax and direct integration with GitHub, which is where I will version control the code.
Node Package Manager (NPM)	This is the project's package manager. This manages all the modules used and stores their names in a config file.
SSD - Storage	Storage will be needed to store every customer & invoice in the database, 5GB will be sufficient to store all of this. SSD has been chosen over HDD as data can be accessed faster due to the lack of moving parts.
Minimum 4GB RAM	This is used in development to host the test server. The end user will also need RAM to allow the web page to render.
A supported Internet browser	Some of the features of ReactJS aren't supported by all browsers. So, an up-to-date, chromium-based browser must be used.
An active internet connection of 512kbps or above	An internet connection is needed to access the website. The user's device will retrieve the HTML of the current page from the web server and render it on their own system's resources.
Laptop / Desktop PC / Tablet	The application is designed to be used in landscape mode on a device with a keyboard. This excludes mobile phones

Success Criteria

- The software needs to integrate with HMRC to submit VAT returns. This is the primary goal of the project so will be extensively tested using their dedicated sandbox server.

- The secondary function of the app is the invoice management software. This will be tested through database queries to make sure that data can be retrieved and processed to be put into a VAT return form.

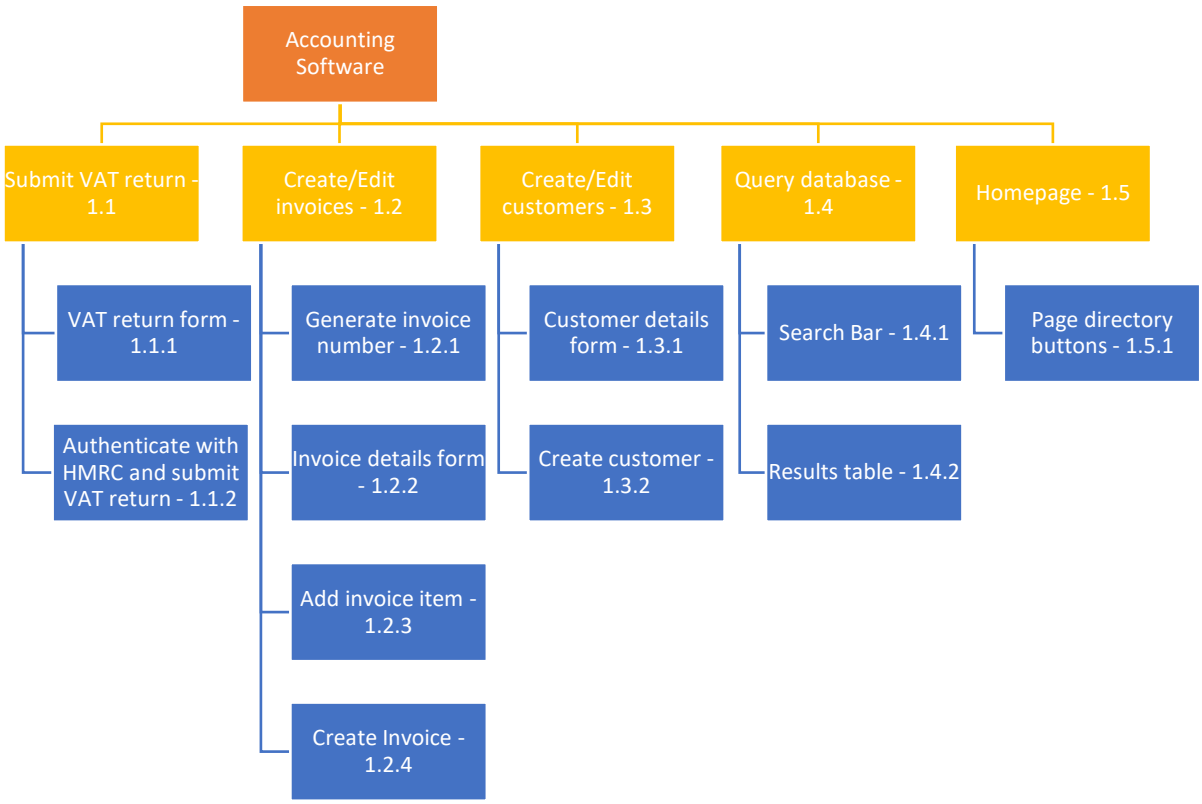
Number		Description	Justification	Measurability
1		Invoices will be added to the database within 5 seconds of clicking submit	This is to ensure that my stakeholders don't get frustrated if they must add lots of invoices	Measured with a stopwatch app
2		Invoice items must show a **real-time** net/vat/total	This makes sure that user verification of prices can be accurate	No visible delay when editing fields
3		Invoices and Customers should load to search table within 3 seconds of page loading	If the user must wait longer, they will question whether data is loading to their screen.	Measured with a stopwatch app
4		Customers should be deleted from the database within 3 seconds of the delete button being pressed	The user might think that the customer hasn't been deleted if they proceed to see it in the table	Measured with a stopwatch app
5		The UI should display on desktop PC, laptop, and landscape iPad screen.	My secondary stakeholder edits invoices on his iPad that he takes to work.	UI should be displayed as intended for each platform. I.e., all the form should be accessible and not running off of the page
6	A	My stakeholders should be able to intuitively create new customers on the system	The whole purpose of the software is ease of use; the program should be able to be used without the need for a guide	My stakeholders will be given a production copy of the program and be told to perform action A, B, C & D. They will be timed with a stopwatch,
	B	My stakeholders should be able to intuitively create new		

		invoices on the system		and I will keep track of how many times they ask for help
	C	My stakeholders should be able to update invoices without a guide.		
	D	My stakeholders should be able to update customer details without a guide.		

Design of the Solution

Decompose the Problem

To begin designing the solution, I must decompose the problem into smaller more manageable ones that I can target and develop one by one to formulate a complete solution in a more efficient workflow.



The diagram above is a detailed hierarchical structure that decomposes the website into its smallest set of components that can each be developed and tested individually. Each of the Yellow headers represents an individual page on the website and the blue headers represent the individual sections of each page. These cannot be divided any further as testing would become inefficient with incomplete and difficult to track outputs. For example, the invoice creation function cannot be tested

without first sanitizing and formatting the input data as firebase uses a proprietary datatype for its dates.

Prototype roadmaps

The first prototype of my solution will consist of creating a basic skeleton for the website and coding the endpoints for the pages. No detail will be added to the pages yet, this will just be a basic skeleton to build on. Laying the foundation of the site's structure early on will give me a framework to follow and better compartmentalise my development of each page individually.

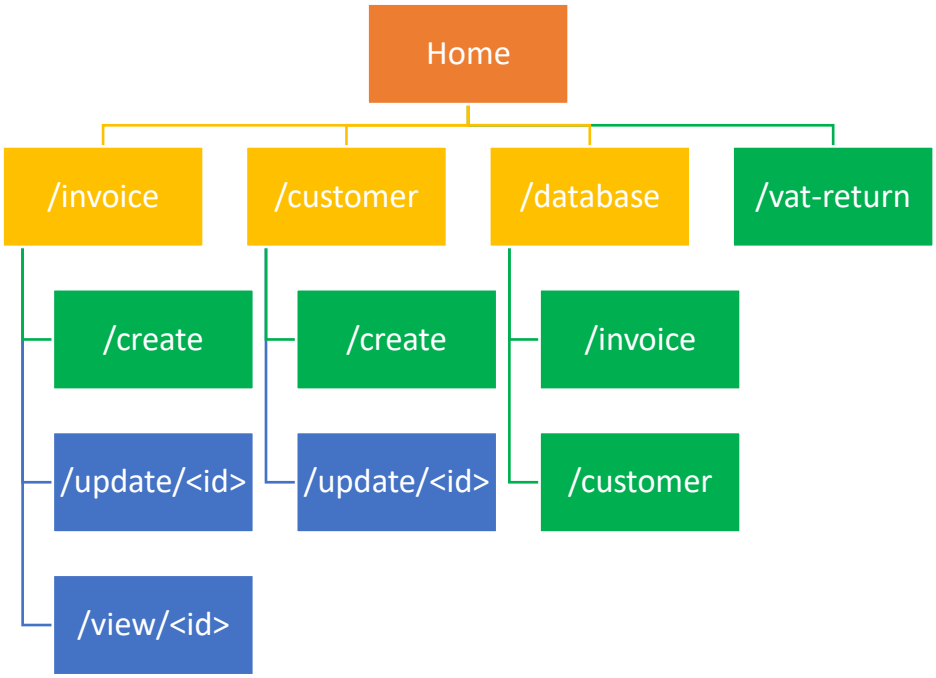
Prototype two will be setting up the visual side of the website with limited to no functionality. All the website's forms will be created but will not do anything. I will also begin to modularise the code, separating reused components into their corresponding files. This will be to get my stakeholder's opinion on the frontend in an earlier stage of iterative development, giving more time to adjust changes they might request.

For prototype three, I will refactor all the functions, forms, and child components of my pages into their own files, making the code easier to read & reducing the number of files with lots of lines. I will organise them into Components, Functions, and Validation. Organisation of code allows for easier addition of features in the future and makes code maintenance less of a workload. I will also implement the backend and integrate the application with my database. This will be a fully functional prototype of the entire final product.

For prototype four. I will implement the VAT return workflow so the user can sign in to their HMRC account and submit VAT returns to HMRC's external API. There will also be a general clean up in terms of the colours used for the User Interface. I will contact my stakeholders and discuss any minor adjustments they wish to make.

Describe the Solution

Site Structure



Root | Groups | Static URLs | Dynamic URLs

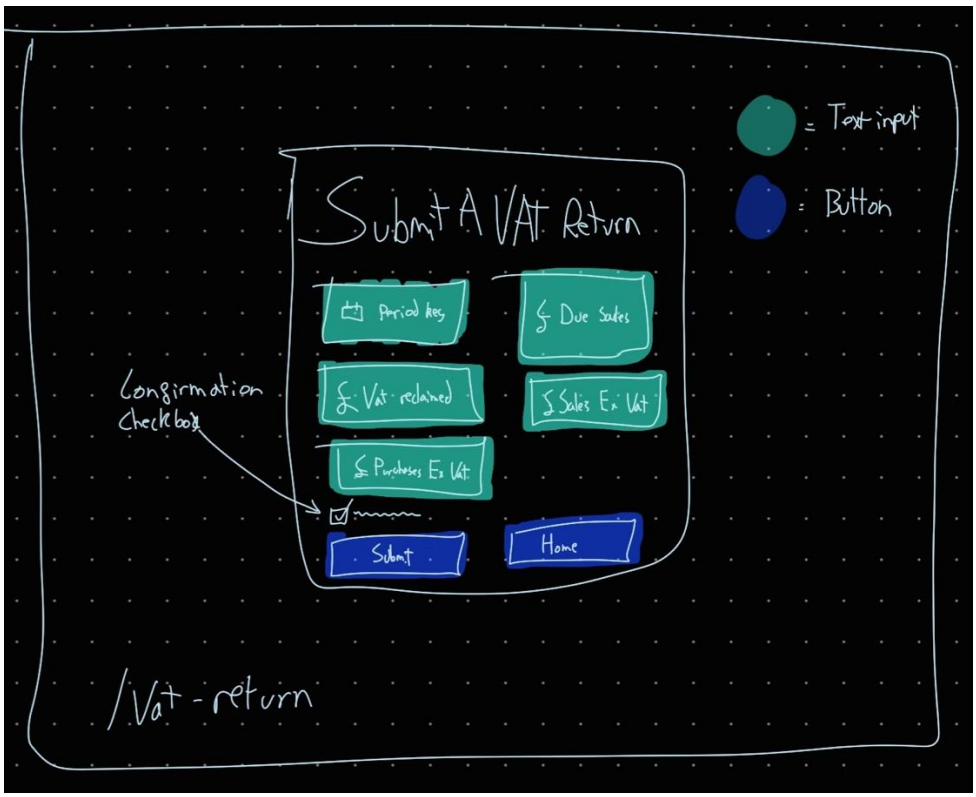
The root is the base URL of the website. It is the page that will be displayed when the user first logs on.

Groups are layouts for similar pages. For example, the create/update/view invoice pages will be the exact same, plus or minus a few details. This means that I can reuse most of the basic functionality code.

Dynamic URLs are pages that take in a parameter – such as an id – and process that data to render the page. Static URLs do not take an input parameter.

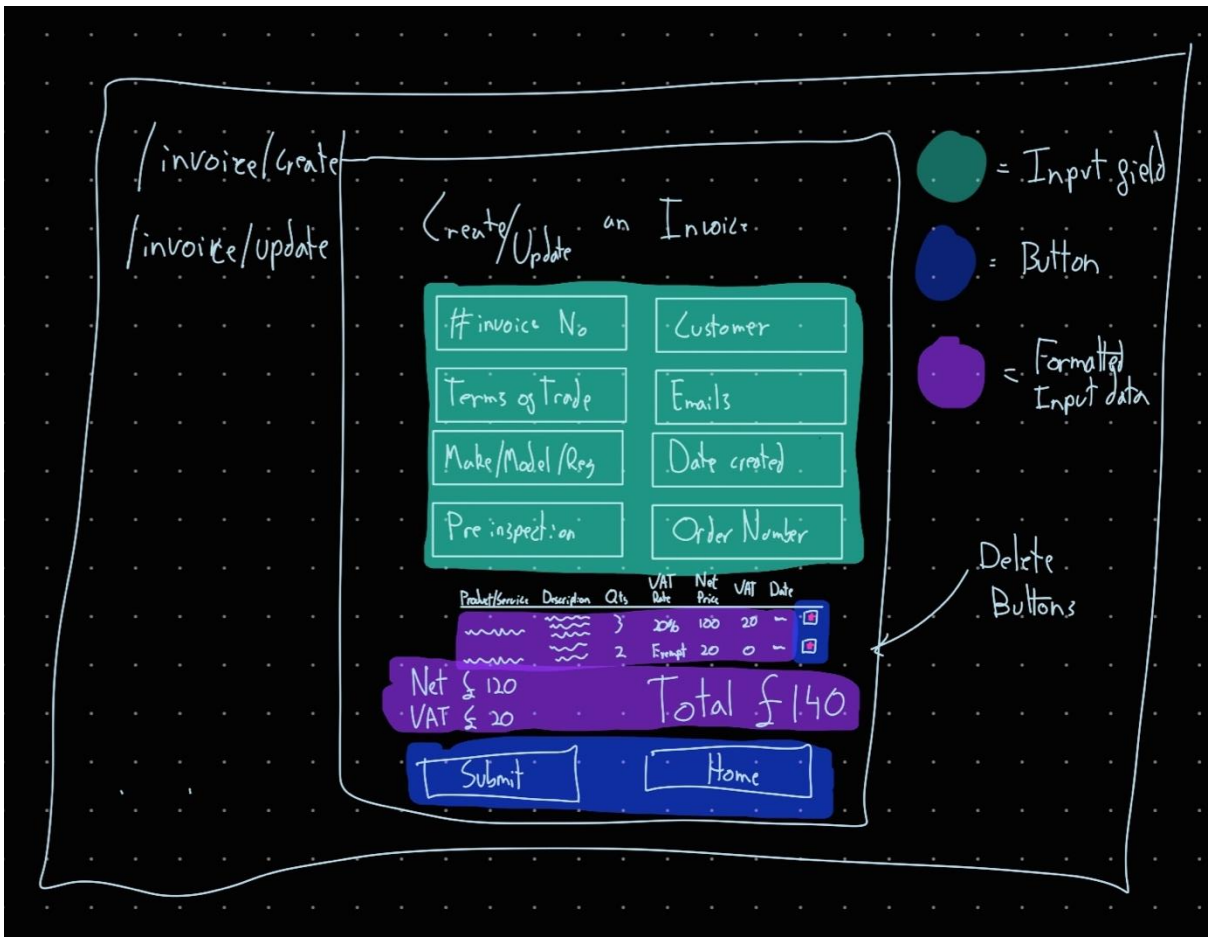
For the layout of individual pages, I sketched a rough wireframe for how I want them to look. Each wireframe has been approved by both my primary and secondary stakeholders, to reduce the amount of change that will be made to the UI design when developing the solution.

/vat-return – 1.1.1



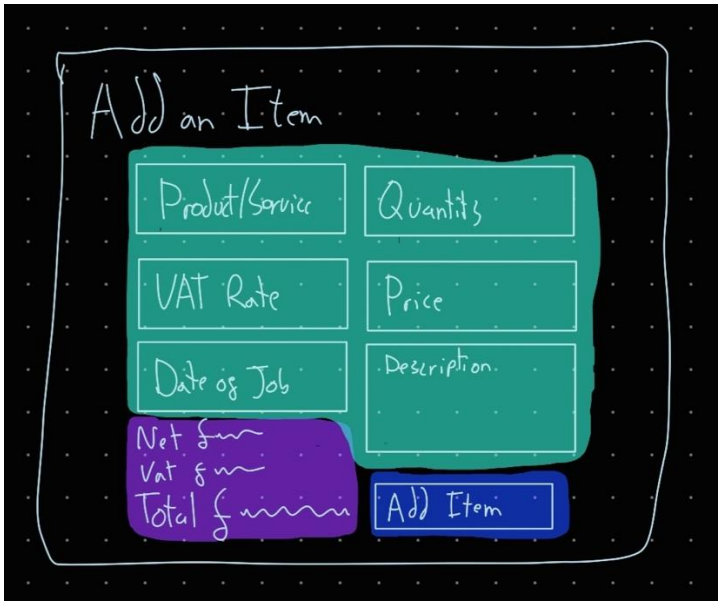
There is a confirmation checkbox, so the user needs to verify the data they have entered before it is sent to HMRC. This is to protect the user from accidentally sending false information to HMRC, so they don't need to contact them asking them to remove/ignore the return.

/Invoice – 1.2.2

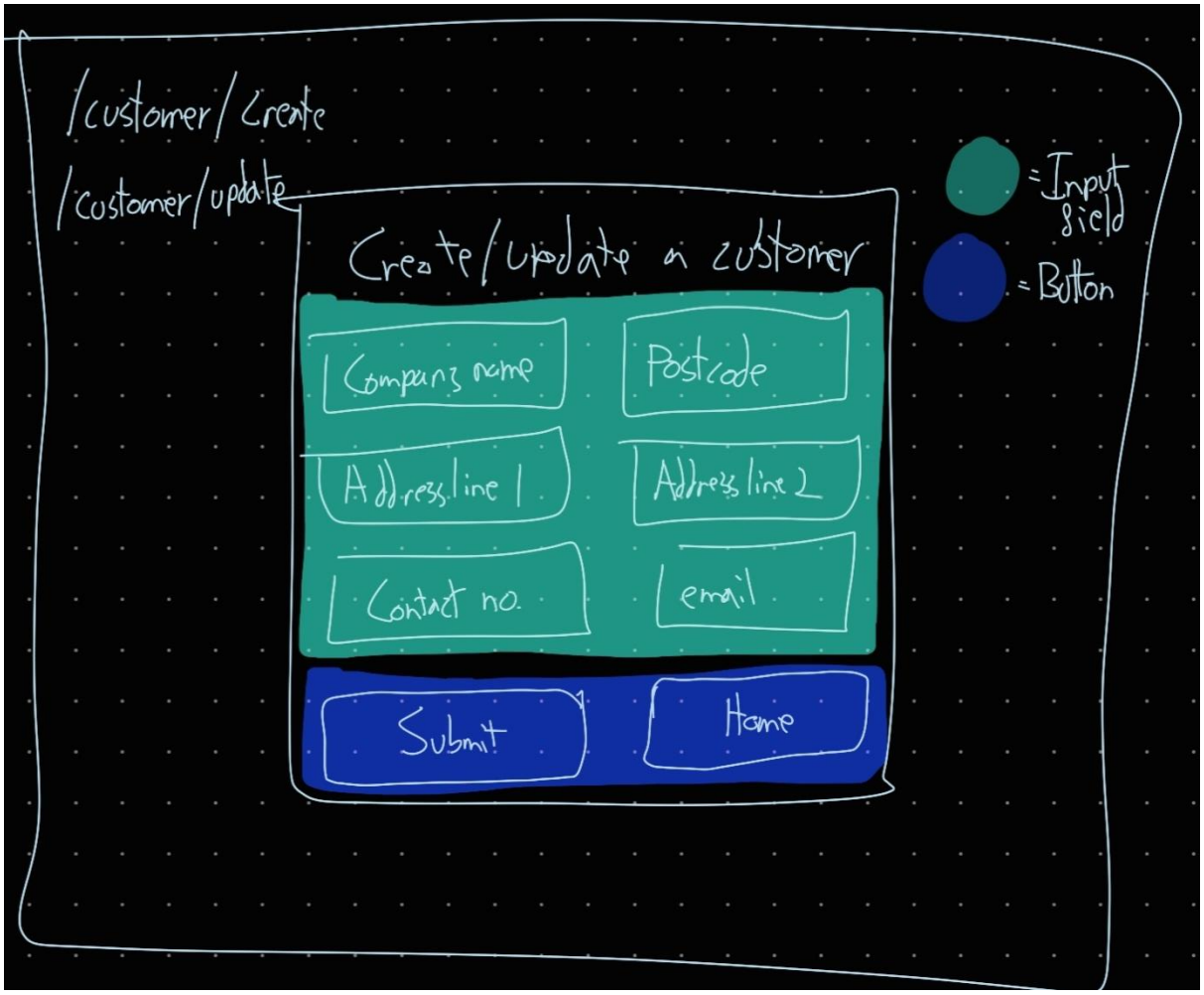


When the user wants to add an invoice item, a popup form will appear as shown below. Data in purple will be processed by the program into an easily readable format but stored in the most efficient way for using it in code. The delete buttons allow the user to remove invoice items if they enter details incorrectly in the popup form.

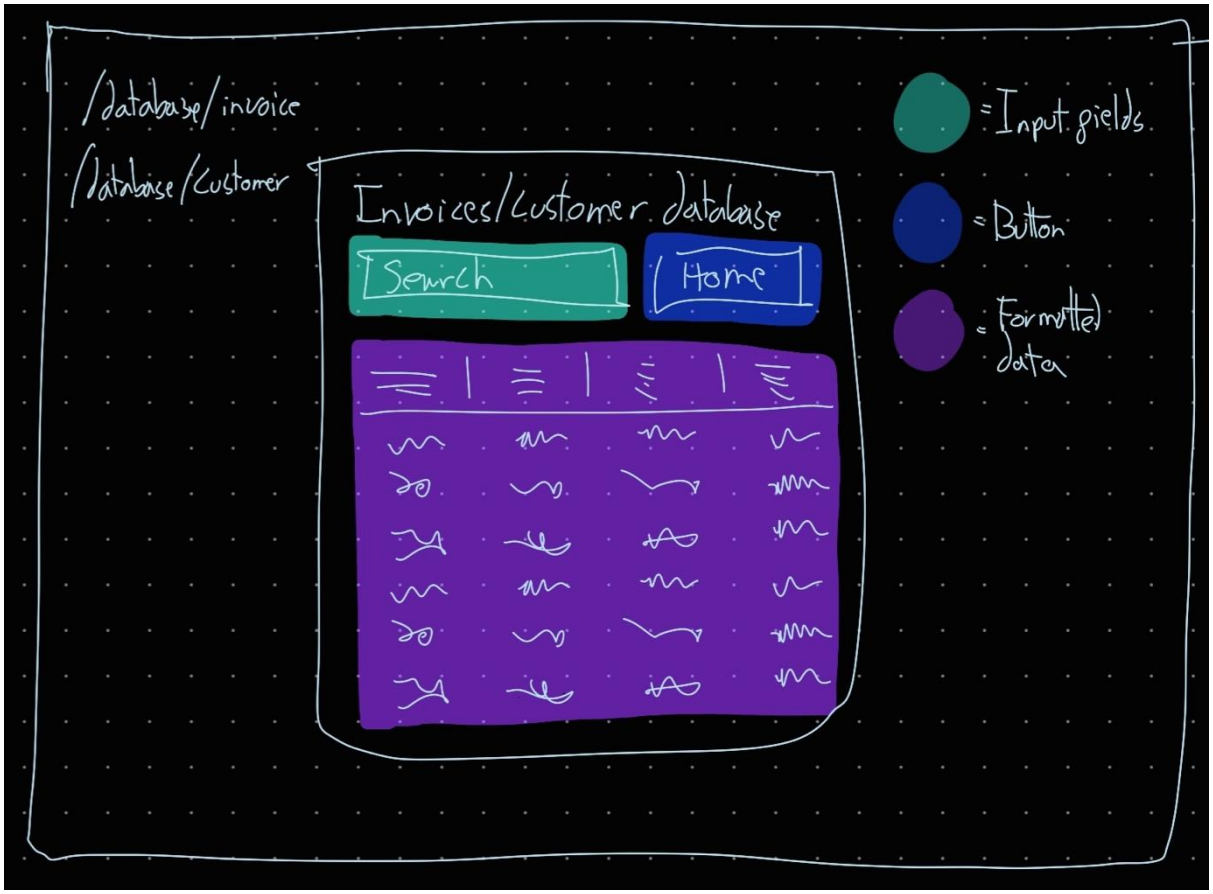
1.2.3



/customer – 1.3.1



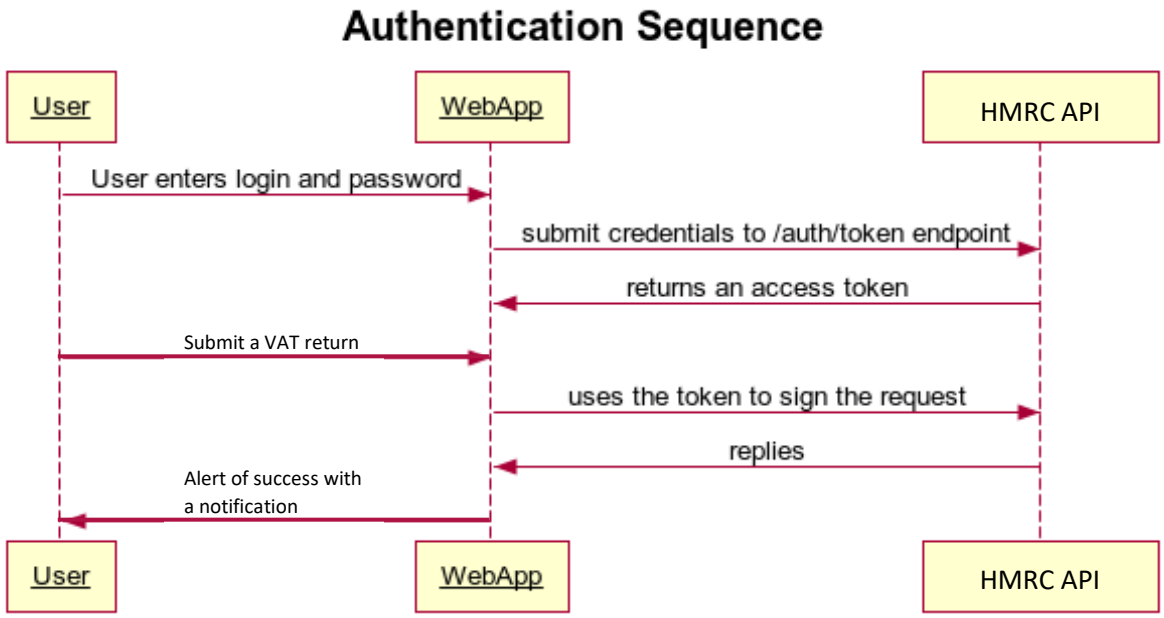
/database – 1.4.1



The search field adapts based on whether the invoice or customer database is being read from. Data in the table will be automatically filtered based on either invoice name or customer name. Only documents that match this criteria will be displayed.

Algorithms

1.1.2



The above diagram is the workflow for authentication with HMRC. Tokens will be stored in the database and associated with a user's unique ID.

```
function submitVatReturn()
    # Gets the VAT data from the form in 1.1.1
    body = getVatData()

    # Makes sure the user has confirmed the data is correct
    if body.finalised == true then
        # Send the data to HMRC
        POST('https://api.service.hmrc.gov.uk/vat/<vat registration
number>/returns', data={ body })
    else
        print("You need to confirm that the data is correct.")
    end if

end function
```

This function gets the VAT data from the form in 1.1.1 and if the user has confirmed the data is correct. It then executes the above authentication workflow, sending the form data off to HMRC.

1.2.1

```
function createInvoiceNumber()
    # Get the number of invoices in the database
    numInvoices = database.invoices().count()
    newNumber = "INV" + (numInvoices+1)
    return newNumber
end function
```

This pseudocode is used to automatically generate a new invoice number when the user wants to create an invoice. Invoice numbers are generated in chronological order (the higher the invoice number, the more recent the invoice). This will allow my stakeholders to keep a consistent cash flow and keep track of when each invoice was created in a more organised manner.

1.4.2

```
procedure filterData(chosenTable, searchFilters)
    if chosenTable == "invoices" then
        data = GET(Invoices)
    else if chosenTable == "customers" then
        data = GET(Customers)
    end if

    filteredData = []

    for item in data
        # Check if each customer meets the user's search filters.
        if meetsFilters(item) == true then
            filteredData.push(item)
        end if
    next for
end procedure
```

The above pseudocode takes all the data from either the invoices or customers table and caches it in the user's browser, so requests don't need to be made to the database every time the search parameters change. The data is then filtered and only relevant records that meet the search requirements are displayed.

Usability Features

Website navigation

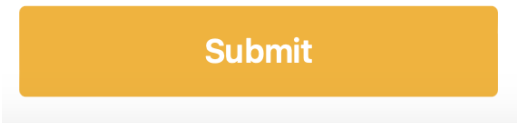
To navigate the website, there will be buttons on the home page that link to each relevant page. There will also be buttons on each other page, redirecting the user back to the homepage. This allows for easy access to all pages on the site and fluent navigation of the system, with the home page being the navigation hub.

Consistent theming

Input fields will have an assigned icon to supplement what their input will be. For example, fields that take currency values will have a pound (£) icon, fields that take a date will have a calendar (🗓️) icon, and fields that take a location/postcode will have a pointer (📍) icon.

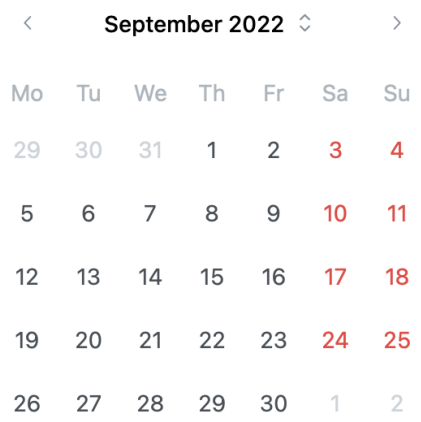
Buttons will be a specific design and colour based on the type of action they perform. This helps the user piece together what each button will do faster than if they were randomly designed. Buttons that redirect the user will be the outlined (no background, coloured text, coloured border) variant for Mantine buttons.

Buttons that perform an action will be the default variant (coloured background, white text)



Buttons that delete data will be outlined red and have a bin (🗑️) icon, to make the user aware that this button will permanently remove data.

Using a pre-made UI library makes consistent sizing of text and buttons significantly easier as a new instance of a component can be rendered to a page with one html tag in React. (e.g. ***<TextInput />***). This also gives me more time to focus on the functionality of the site rather than styling with CSS which wouldn't be feasible in the given time frame considering the complexity of some of the components (such as the Calendar system as shown below)



Notifications System

MantineUI has a built in system for displaying notifications to the user. I can use these to give the user information about the page that they are on. These are better than console.log() as they appear over the actual website rather than in the console output. These will be used to alert of validation errors and when something is created/updated in the database.

Variables

Customer Variables

Variable	Justification/Description	Validation	Justification
companyName	Used to refer to the customer throughout the UI	N/A	N/A
postcode	For my stakeholder's user rather than the system's. Easier to put in a satnav than full address	Must be 8 Characters or less	The UK postcode format is 3 or 4 characters followed by an optional space and 3 more characters
address1	Stores the first line of the address	N/A	N/A
address2	Stores the second line of the address	N/A	N/A
contactNo	For my stakeholders to contact customers on any issues	Must only contain numbers and spaces	Corresponds to the phone number of the customer so must be in the correct form
email	Used by the system to autocomplete the form in an invoice	Must be in the correct email form (e.g., name@domain.com)	So my stakeholders can copy/paste the email straight into external documents without worrying about formatting

Invoice Variables

Variable	Justification/Description	Validation	Justification
invoiceNumber	Used as the primary key to uniquely identify all invoices. Generated in ascending numerical order.	Must be in the form INVXXXX where X represents a number from 0-9	To maintain a standardised naming system so when invoices are sent to customers, they can be organised by them.
customer	Stores a reference to the invoice's customer. This allows for autocompletion of the email field if the customer is already in the database	The customer is checked every time the edit invoice page is loaded to see if they are still in the database	To notify my stakeholders that the customer used in the invoice can no longer be used to create invoices. But their email will still be stored on all previous invoices
termsOfTrade	To tell the app how long a customer must pay an invoice when the system is calculating a due date	N/A	The field is a selection field with all items pre-defined
dateCreated	Tells my stakeholders when the invoice was created. Used by the app when calculating a due date	The date must be on or before the current date, never after	An invoice cannot have been created 3 days from now
email	Stores the customer's email address for ease of access by my stakeholders when viewing invoices	Must be in the correct email form (e.g., name@domain.com)	So my stakeholders can copy/paste the email straight into external documents without worrying about formatting
makeModelReg	Stores the make/model/registration of the bus used on the job. For my stakeholder's organisation purposes	N/A	N/A
preInspection	Any damage found on the vehicle before the job has started. To cover my stakeholders in case damage is found after the job.	N/A	N/A
orderNo	The order/invoice number of parts purchased for the job	N/A	numbers can be in any format as my stakeholder's buy items from various suppliers who all use different naming conventions.
datePaid	To store the date that the invoice was paid by the customer	Cannot be set to a date after the current date or before the invoice was created.	The customer cannot have paid for a job before they have received the invoice for it.

Invoice Item Variables

Variable	Justification/Description	Validation	Justification
service	Brief title of the individual service supplied to the customer	Must be no more than 30 characters	The title is supposed to be brief, more detail will be proved in the item description
quantity	Stores the number of times the service has been supplied, for example, in the case of stone chip repairs	Must be an integer above 0	There must have been at least one time the service has been used otherwise it is not needed on the invoice
vatRate	The VAT rate for individual invoice items can be different so each one will store their own rate	N/A	vatRate is a selection field so all inputs are pre-defined
price	Stores the price for one unit of the service	Must be a positive decimal	Price of an object cannot be negative
date	Used to store the date that the service was carried out. Sometimes different to the date the invoice was created	Can be set to any date before the invoice's creation date	The invoice cannot be created before the service has been carried out
description	A more in-detail description of what the service was that was carried out.	Must be no more than 100 characters	The description cannot be a full paragraph as this would break the formatting of the webpage.

The project's backend will query from a NoSQL database. For 1.1.1, The data will be validated client side before being sent to the server to reduce server load.

For 1.1.2, the external VAT API will be authorized and managed on the backend as doing this on the frontend is insecure and results in clashes with the Cross Origin Resource Sharing (CORS) settings of the external API.

All inputs in each form will pass through a verification of both length and format. Emails will have their own validation (e.g., email@example.com) and the same for mobile numbers (only contains numbers and spaces). If these requirements aren't met, the form will not submit.

Describe the Approach to Testing

In-Development Test Plan

I will test each function individually before integrating it into the main project. I will also test the whole program at regular intervals with custom test data. I will also make a checklist of each component in the site to use after development has concluded.

Prototype 1

Function/Component to test	Justification	Test Data
<App />	To ensure that the frontend of the website will run.	No specific data, the program will return an error if the component doesn't work
URL routes for the site	To ensure that the routes the user should be able to access return the correct pages.	/ /vat-return /customers/update/<CUSTOMER_ID> /customers/create /invoices/create /invoices/update/<INVOICE_ID> /database/invoices /database/customers

Prototype 2

Test No.	Function/Component to test	Justification	Test Data	Expected Result
1	addItem (in the <CreateUpdateInvoice /> component)	To insure that items can actually be added to the invoice	"Add Item" button pressed User Has filled out the invoice item form	Invoice item modal should appear. Add the item to the invoiceItems array and update the Net/Vat/Total for the invoice

Prototype 3

Test No.	Function/Component to test	Justification	Test Data	Expected Result
1	<SignOutButton />	To ensure the user can sign out of their account on shared devices	No input data, the button is pressed whilst the user is signed in	The user is signed out and sees the SignedOut page. A notification is also displayed.
2	validateEmail()	To make sure that emails are in the correct format for copy-pasting	"email@example.com" "testCompany@gmail.com" "notAnEmail"	True True False
3	validateMobile()	To make sure that mobile numbers are in the correct	"07123 123 122" "01924 123452" "fakeMobileNumber"	True True False

		format for copy-pasting		
4	validateInvoiceNumber()	To make sure that invoice numbers are in the correct format for easy organisation	INV0001 INV0125 INV001231 INV01	True True False False
5	getCustomerInfo()	To make sure that customer data can be retrieved from the database.	Input a valid customerID Input an invalid customerID	Returns the data for that customer Returns the Boolean “false”
6	getInvoiceInfo()	To make sure that invoice data can be retrieved from the database	Input a valid Invoice number Input an invalid invoice number	6a - Returns the data for that invoice 6b - Returns the Boolean “false”
7	updateCustomer()	To ensure that existing customer data can be updated if things change or are inputted incorrectly	Validation of data isn’t needed for this function as all inputted data will already be validated. Input valid customerData and a customerID	Display a notification to the user and update the database to show the updated customer
8	createCustomer()	To ensure that new customers can be created with their data being stored in the database’s correct collection	Validation of data isn’t needed for this function as all inputted data will already be validated. Input valid customerData	Display a notification to the user and create a new customer document in the database’s “Customers” collection
9	updateInvoice()	To ensure that existing invoice data can be updated if things change or are inputted incorrectly	Validation of data isn’t needed for this function as all inputted data will already be validated. Input valid invoiceData and an invoice number	Display a notification to the user and update the database to show the updated invoice

10	createInvoice()	To ensure that new invoices can be created with their data being stored in the database's correct collection	Validation of data isn't needed for this function as all inputted data will already be validated. Input valid invoiceData	Display a notification to the user and create a new invoice document in the database's "Invoices" collection
11	generateInvoiceNumber()	To make sure that invoice numbers automatically generate, reducing the user workload and increasing ease of organisation.	No data is inputted, the number is determined based on the number of existing invoices	Return the next generated invoice number in the correct format ("INVxxxx")
12	populateCustomerDetails()	To make sure		
13	<HomeButton />	To make sure the user can navigate the website as the home page is a hub of all links to other pages.	No data inputted other than the button being pressed on each page of the website	Redirect the user to the home page.
14	<BackButton />	To make sure the user can navigate the website as the home page is a hub of all links to other pages.	The "to" prop will store the url of the webpage the user should be directed to. This is in the code rather than user inputted data so no validation is needed.	Redirect the user to the page marked in the "to" prop

15	<EditButton />	To make sure the user can navigate between invoice/customer update pages and the database search pages.	The "table" and "id" props tell the button which URL to redirect the user to, for example: {table: "customers", id:"zx5cfg6dehh" } {table: "invoices", id:"INV0007"}	Redirect the user to ***/customer/update/ zx5cfg6dehh*** Redirect the user to ***/invoice/update/INV0007***

Post-Development Test Plan

I will deploy the final build of the website to firebase and test each individual component of the site in accordance with my checklist that I made during development. As well as this – my stakeholders will be actively using the web app for their business after I have finished development. I will remain in close contact with them in case they have any questions/found bugs/minor tweaks they want.

Test Number		Explanation	Test Data	Expected Result	Justification
1		When creating a customer, there is a maximum character limit for each field	No specific data inputted, just spam letters into each field to test how many characters each field allows.	After a certain length, each field will stop editing the input as the maximum length has been reached	To validate that the data will be displayed correctly when sent back to the user. Longer inputs would mess with the pages formatting.
2	**A**	When submitting a customer creation form, the client puts the inputted email address and mobile number through validation function to check that they are in	**For email field:** testCustomer@gmail.com	PASS	To make sure that the validation for customers works as intended
	B		customerEmail1@test.co.uk	PASS	
	C		"notAnEmail"	Fail Notification	
	D		**For mobile field:** "01924 251 222"	PASS	

	E	the correct format.	“0800 122 223”	PASS	
	F		“ ” (empty form)	Fail Notification	
3		Test the entire workflow of creating a customer. With the software being used as intended	No specific data inputted, just start on the home page and work through creating a customer on the system from start to finish.	The customer will be created and the user will receive a notification alerting the customer has been created.	This will contribute to the success of criteria 6A (see analysis)
4		Test the entire workflow for creating an invoice. With the software being used as intended	No specific data inputted, just start on the home page and work through creating a customer on the system from start to finish.	The invoice will be created and the user will receive a notification alerting them of this	This will contribute to the success of criteria 6B (see analysis)
5		Test the process of logging in and out if the software using a google account.	No specific data inputted, just start on the logged out page and sign in with a Gmail account, then sign out again	The user will see the home page when they log in and get a notification when they sign out	To make sure that all users of the webpage are recorded so my stakeholders have a clear view of who is accessing the site and can block users that aren’t authorized.
6	**A**	Test the entire workflow of updating / deleting a customer	The user will start on the homepage and work through updating a specific field on a customer.	The user will get a notification to alert them for each stage	This will contribute to the success of criteria 6D (see analysis)
	B		The user will delete a customer.		
7		Test the entire workflow of updating an invoice	No specific data inputted. The user will start on the homepage and work through updating an invoice’s fields and paid status.	The user will receive a notification that they have successfully updated an invoice	This will contribute to the success of criteria 6C (see analysis)
8		Test that invoices behave correctly when their respective	No specific data inputted. The user will view an invoice whose customer is no longer in the database.	The program will display “Deleted” in the customer	This test will make sure that the program will

	customer has been deleted		input box and the user will get a warning notification.	handle customer-less invoices and alert the user that they can no longer use that customer for future invoices.

Developing the Solution

Iterative Development Process

I used GitHub to version control my web app; the repo can be found here https://github.com/daniel-555/A2-CS-Project/. As a direct result of this, each section of my development are individual commits.

Prototype 1

https://github.com/daniel-555/A2-CS-Project/tree/Prototype1

For this prototype, I created the basic layout of the website - directing each of the routes to their respective pages.

App.jsx

27 lines (24 sloc) | 905 Bytes — Raw | Blame

```jsx
import { BrowserRouter, Routes, Route } from "react-router-dom";
import CreateUpdateCustomer from "./Pages/CreateUpdateCustomer";
import CreateUpdateInvoice from "./Pages/CreateUpdateInvoice";
import DatabaseQuery from "./Pages/DatabaseQuery";
import HomePage from "./Pages/HomePage";
import VATReturn from "./Pages/VATReturn";

const App = () => {
        return (
                <BrowserRouter>
                        <Routes>
                                <Route path="/" element={<HomePage />} />
                                <Route path="/vat-return" element={<VATReturn />} />
                                <Route
                                        path="/customer/:action"
                                        element={<CreateUpdateCustomer />}
                                />
                                <Route path="/invoice/:action" element={<CreateUpdateInvoice />} />
                                <Route path="/database/:collection" element={<DatabaseQuery />} />
                        </Routes>
                </BrowserRouter>
        );
};

// This file is the base of the whole application. It routes all of the different pages to the code each one should run.

export default App;
```

To route the pages, I used the library react-router-dom, which allows me to assign different paths to specific react components (i.e., the page files).

I also created blank components for each page to test that the routing was working properly.

The :action and :collection tags allow for parameters to be added to the URL which are read by the corresponding components using the useParams() hook. This allows: the customer & invoice pages to differentiate between creating and updating; the database query page to differentiate between database collections.

CreateUpdateCustomer.jsx

11 lines (8 sloc) | 369 Bytes — Raw | Blame

```jsx
import { Title } from "@mantine/core";
import { useParams } from "react-router-dom";

const CreateUpdateCustomer = () => {
        const { action } = useParams();
        return <Title order={1}>This is the {action} customer page</Title>;
};

// This component will allow the user to create a new customer or edit the details of an existing one

export default CreateUpdateCustomer;
```

/customer/create

/customer/update

CreateUpdateInvoice.jsx

11 lines (8 sloc) | 356 Bytes — Raw | Blame

```jsx
import { Title } from "@mantine/core";
import { useParams } from "react-router-dom";

const CreateUpdateInvoice = () => {
        const { action } = useParams();
        return <Title order={1}>This is the {action} invoice page</Title>;
};

// This page will allow the user to create a new invoice or edit details on an existing one

export default CreateUpdateInvoice;
```

/invoice/create

This is the create invoice page

/invoice/update

This is the update invoice page

DatabaseQuery.jsx

12 lines (9 sloc) | 421 Bytes — Raw | Blame

```jsx
import { Title } from "@mantine/core";
import { useParams } from "react-router-dom";

const DatabaseQuery = () => {
        const { collection } = useParams();
        return <Title order={1}>This is the {collection} query page</Title>;
};

// This component will allow the user to query both the customers and invoices database.
// The results will be shown in a table with all relevant data displayed.

export default DatabaseQuery;
```

/database/customers

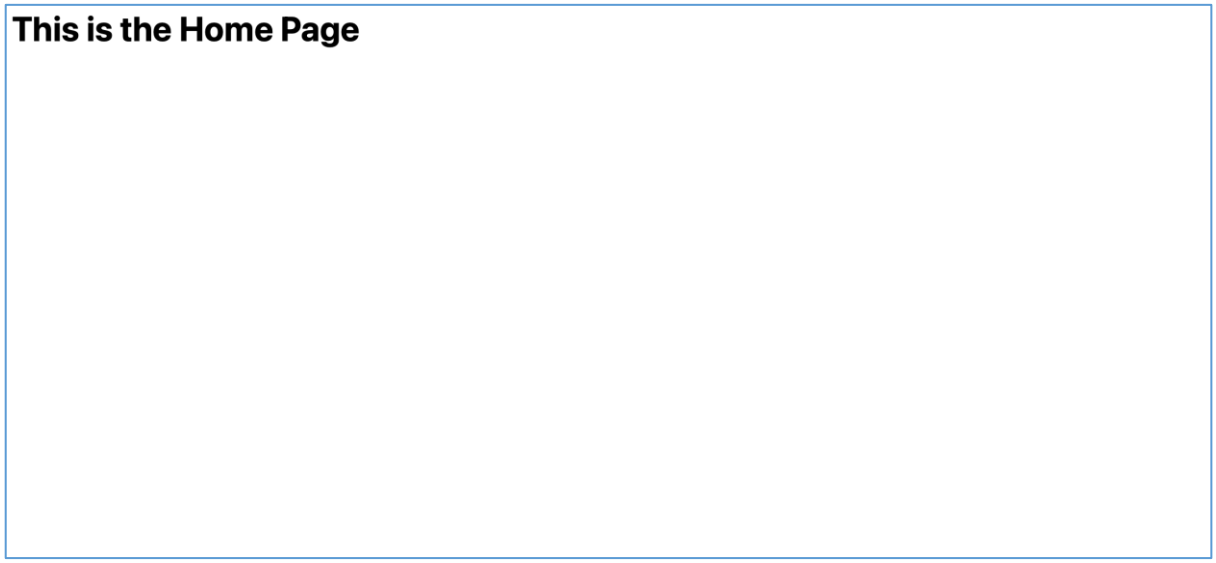
/database/invoices

HomePage.jsx

10 lines (7 sloc) | 302 Bytes — Raw | Blame

```jsx
import { Title } from "@mantine/core";

const HomePage = () => {
        return <Title order={1}>This is the Home Page</Title>;
};

// This is the website's home page.
// The app's title and my stakeholder's business logo will be displayed here, along with links to all other pages.

export default HomePage;
```

/

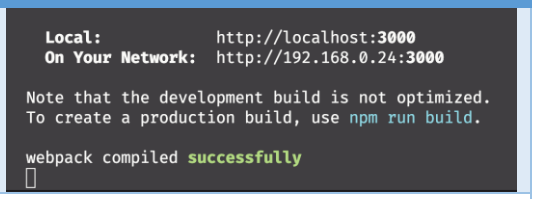


VATReturn.jsx

9 lines (6 sloc) | 290 Bytes — Raw | Blame

```jsx
import { Title } from "@mantine/core";

const VATReturn = () => {
        return <Title order={1}>This is the VAT return form page.</Title>;
};

// This page will allow the user to submit a VAT return to HMRC, putting them through the necessary authentication to do so.

export default VATReturn;
```

/vat-return

Test	Pass / Fail	Evidence
Running the program to ensure no errors are returned	PASS	
Testing all URLs to make sure they return the correct pages	PASS	All the page screenshots above show that pages are functioning correctly

Prototype 2

https://github.com/daniel-555/A2-CS-Project/tree/Prototype2

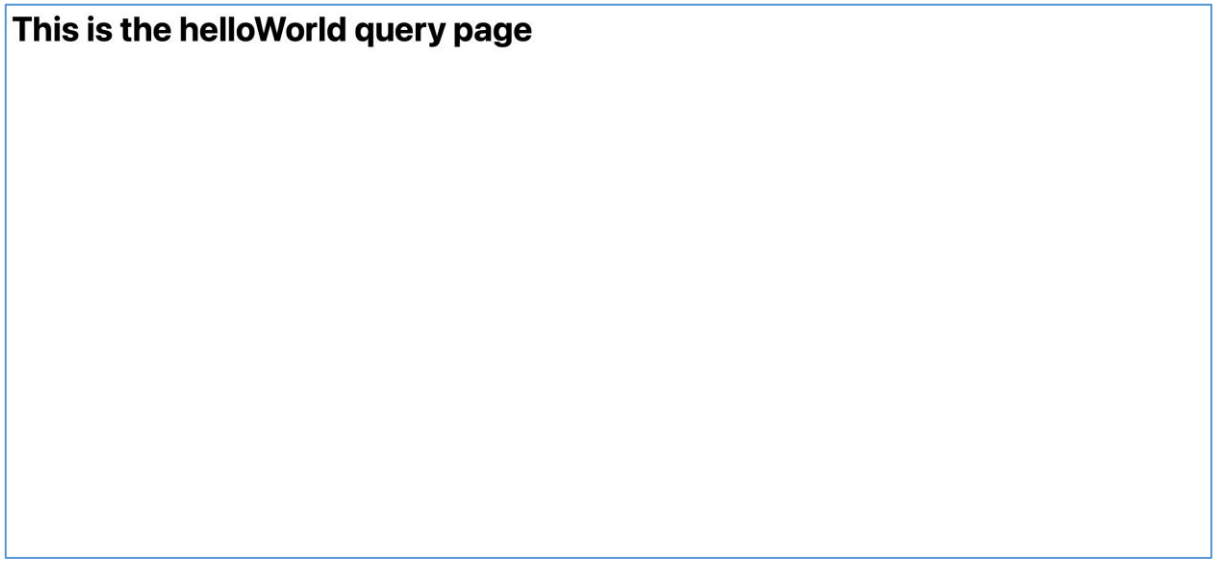
Refactoring of the routing system

Whilst testing the routing system for the website, I came across a major flaw with the way I was handling the invoice & customer create/update pages, along with the database query pages. The way I laid out the *:action* parameter allowed the user to enter anything into their URL and the page would interpret that into the title (examples below).

/invoice/testPhrase

This is the testPhrase invoice page

/database/helloWorld



To resolve this issue, I refactored the routing system to only allow the specific endpoints I intend the user to access. I did this through nesting routes within each other and adding the collection/action parameters as props for the components instead.

```
import { MantineProvider } from "@mantine/core";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import CreateUpdateCustomer from "./Pages/CreateUpdateCustomer";
import CreateUpdateInvoice from "./Pages/CreateUpdateInvoice";
import DatabaseQuery from "./Pages/DatabaseQuery";
import HomePage from "./Pages/HomePage";
import VATReturn from "./Pages/VATReturn";

const App = () => {
        return (
                <MantineProvider>
                        <BrowserRouter>
                                <Routes>
                                        <Route path="/" element={<HomePage />} />
                                        <Route path="/vat-return" element={<VATReturn />} />
                                        <Route path="/customer">
                                                <Route
                                                        path="create"
                                                        element={<CreateUpdateCustomer action="create" />}
                                                />
                                                <Route
                                                        path="update/:customer"
                                                        element={<CreateUpdateCustomer action="update" />}
                                                />
                                        </Route>
                                        <Route path="/invoice">
                                                <Route
                                                        path="create"
                                                        element={<CreateUpdateInvoice action="create" />}
                                                />
                                                <Route
                                                        path="update"
                                                        element={<CreateUpdateInvoice action="/update" />}
                                                />
                                        </Route>
                                        <Route path="/database">
                                                <Route
                                                        path="customers"
                                                        element={<DatabaseQuery collection="customers" />}
                                                />
                                                <Route
                                                        path="invoices"
                                                        element={<DatabaseQuery collection="invoices" />}
                                                />
                                        </Route>
                                </Routes>
                        </BrowserRouter>
                </MantineProvider>
        );
};

// This file is the base of the whole application. It routes all of the different pages to the code each one should run.

export default App;
```

UI of the VAT return form page completed

https://github.com/daniel-555/A2-CS-Project/tree/68830a71ecd3554f981bda07e28586e312a7ce2f

My next course of action was to complete the UI for the VAT return form, I used components from the Mantine UI library and my own colour palette in setting out the page.

VATReturn.jsx

62 lines (60 sloc) | 1.93 KB — Raw | Blame

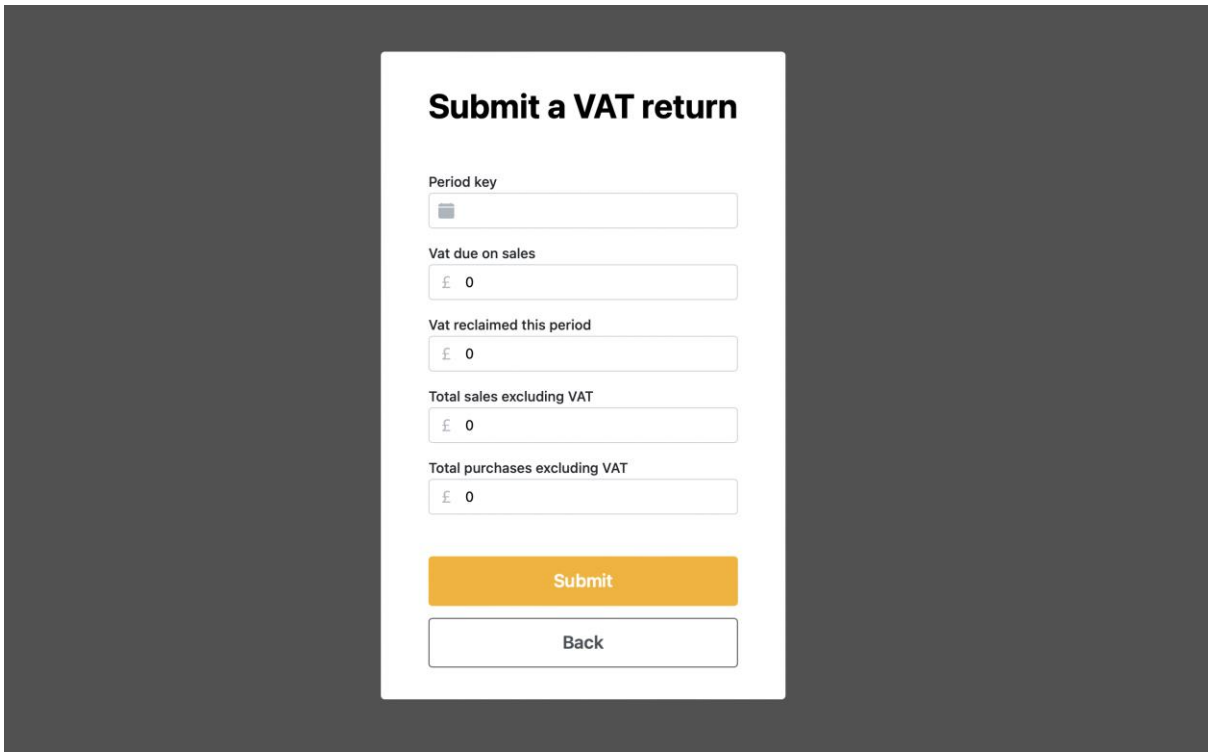
```jsx
import { Card, SimpleGrid, TextInput, Title, Button } from "@mantine/core";
import { useState } from "react";
import { BsFillCalendarFill, BsCurrencyPound } from "react-icons/bs";
const VATReturn = () => {
        const [periodKey, setPeriodKey] = useState("");
        const [vatDueSales, setVatDueSales] = useState(0);
        const [vatReclaimed, setVatReclaimed] = useState(0);
        const [totalSalesExVat, setTotalSalesExVat] = useState(0);
        const [totalPurchasesExVat, setTotalPurchasesExVat] = useState(0);
        const [finalised, setFinalised] = useState(false);

        return (
                <Card className="card center">
                        <form>
                                <SimpleGrid cols={1} spacing="sm">
                                        <Title order={1}>Submit a VAT return</Title>
                                        <br />
                                        <TextInput
                                                label="Period key"
                                                icon={<BsFillCalendarFill />}
                                                value={periodKey}
                                                onChange={(e) => setPeriodKey(e.target.value)}
                                        />
                                        <TextInput
                                                label="Vat due on sales"
                                                icon={<BsCurrencyPound />}
                                                value={vatDueSales}
                                                onChange={(e) => setVatDueSales(e.target.value)}
                                        />
                                        <TextInput
                                                label="Vat reclaimed this period"
                                                icon={<BsCurrencyPound />}
                                                value={vatReclaimed}
                                                onChange={(e) => setVatReclaimed(e.target.value)}
                                        />
                                        <TextInput
                                                label="Total sales excluding VAT"
                                                icon={<BsCurrencyPound />}
                                                value={totalSalesExVat}
                                                onChange={(e) => setTotalSalesExVat(e.target.value)}
                                        />
```

c

```jsx
                                        <TextInput
                                                label="Total purchases excluding VAT"
                                                icon={<BsCurrencyPound />}
                                                value={totalPurchasesExVat}
                                                onChange={(e) => setTotalPurchasesExVat(e.target.value)}
                                        />
                                        <br />
                                        <Button fullWidth size="lg" color="yellow.6">
                                                Submit
                                        </Button>
                                        <Button fullWidth size="lg" color="gray.7" variant="outline">
                                                Back
                                        </Button>
                                </SimpleGrid>
                        </form>
                </Card>
        );
};
// This page will allow the user to submit a VAT return to HMRC, putting them through the necessary authentication to do so.

export default VATReturn;
```

All other values for the VAT return can be calculated when the request is sent, rather than have the user calculate and input them. At this stage in development, none of the buttons have functionality and there is no logic behind any of the text inputs, I will complete all the website's UI first because then it is easier to make the site look consistent in design across pages.

/vat-return



Partially completed UI for the create/update invoice page

https://github.com/daniel-555/A2-CS-Project/tree/48cad6d7434d6f1a2575ddfa47ffa3d26a99d937

I then completed the first part of the UI for the create/update invoice page. I finished this up to just before the billable items section, which will require a bit more work to get done so will be done in its own section to better compartmentalise development. I also added a parameter to the update route (:invoice) which will allow me to edit a specific invoice by entering the number into the browser.

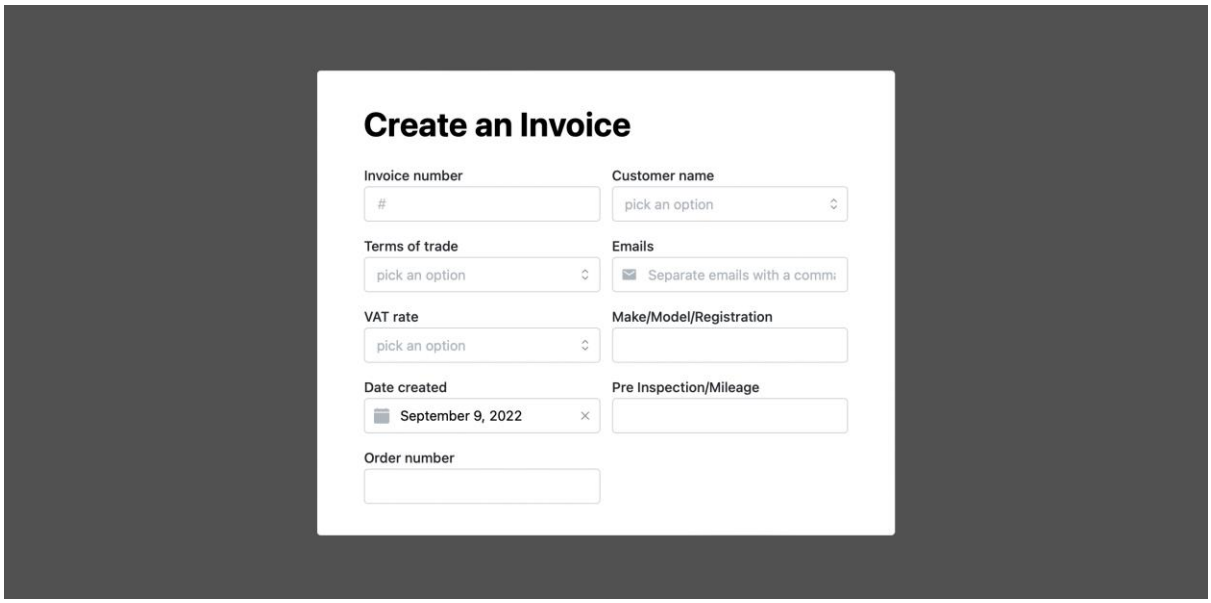
App.jsx

CreateUpdateInvoice.jsx

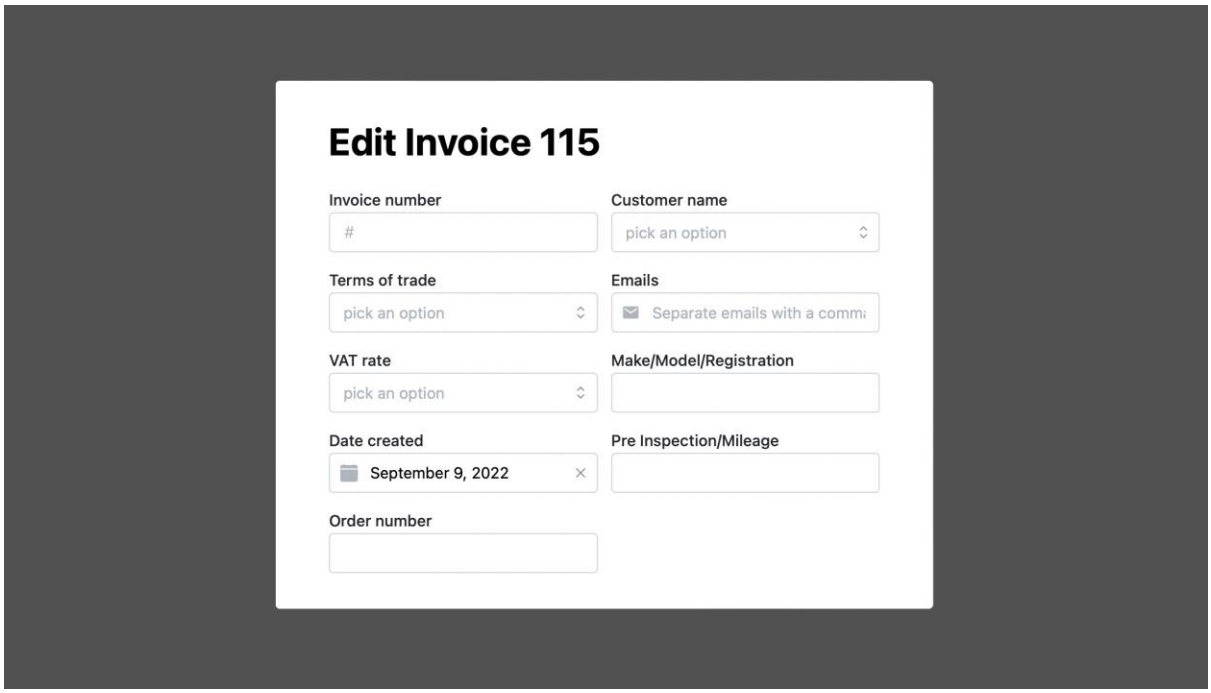
```
                                        : `Edit Invoice ${invoice}`}
                                </Title>
                                <br />
                                <SimpleGrid cols={2} spacing="sm">
                                        <TextInput
                                                label="Invoice number"
                                                icon={<BsHash />}
                                                value={invoiceNumber}
                                                onChange={(e) => setInvoiceNumber(e.target.value)}
                                        />
                                        <Select
                                                label="Customer name"
                                                placeholder="pick an option"
                                                searchable
                                                nothingFound="No customers with that name"
                                                data={customerData}
                                                value={selectedCustomer}
                                                onChange={setSelectedCustomer}
                                        />
                                        <Select
                                                label="Terms of trade"
                                                placeholder="pick an option"
                                                data={termsOfTradeData}
                                                value={termsOfTrade}
                                                onChange={setTermsOfTrade}
                                        />
                                        <TextInput
                                                label="Emails"
                                                icon={<IoMdMail />}
                                                placeholder="Separate emails with a comma"
                                                value={emails}
                                                onChange={(e) => setEmails(e.target.value)}
                                        />
                                        <Select
                                                label="VAT rate"
                                                placeholder="pick an option"
                                                data={vatRateData}
                                                value={vatRate}
                                                onChange={setVatRate}
                                        />
                                        <TextInput
                                                label="Make/Model/Registration"
                                                value={makeModelReg}
                                                onChange={(e) => setMakeModelReg(e.target.value)}
```

```
                                        />
                                        <DatePicker
                                                label="Date created"
                                                icon={<BsFillCalendarFill />}
                                                value={dateCreated}
                                                onChange={setDateCreated}
                                        />
                                        <TextInput
                                                label="Pre Inspection/Mileage"
                                                value={preInspection}
                                                onChange={(e) => setPreInspection(e.target.value)}
                                        />
                                        <TextInput
                                                label="Order number"
                                                value={orderNumber}
                                                onChange={(e) => setOrderNumber(e.target.value)}
                                        />
                                </SimpleGrid>
                        </form>
                </Card>
        );
};

// This page will allow the user to create a new invoice or edit details on an existing one

export default CreateUpdateInvoice;
```

/invoice/create



/invoice/update/115



Another thing I noticed is that I was creating the invoice form in the file ***CreateUpdateCustomer.jsx*** instead of ***CreateUpdateInvoice.jsx***. I discovered this by testing each endpoint on the website and when I did /customer/create, the create invoice form popped up. To fix this, I just moved the code to the correct file (***CreateUpdateInvoice.jsx***).

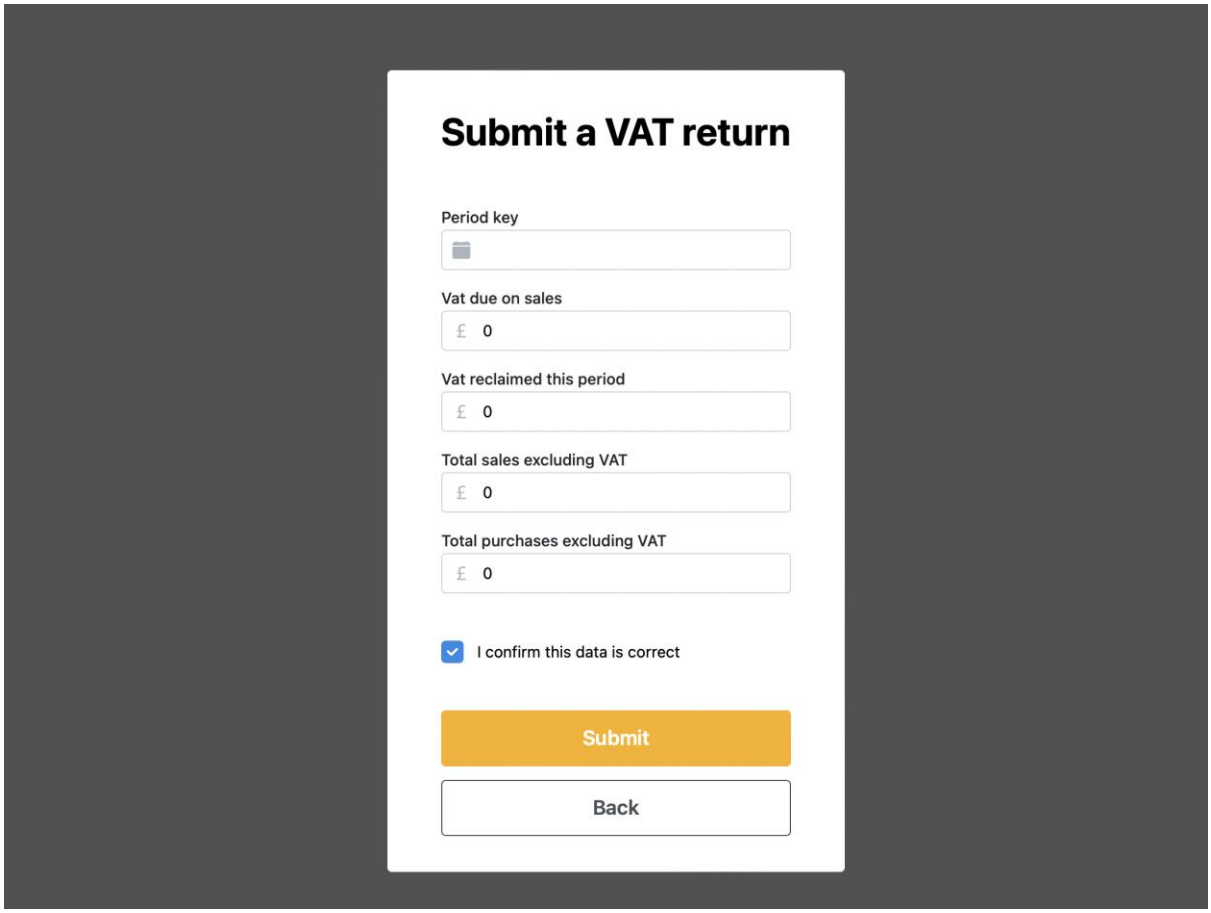
Added checkbox to Vat return form

In this commit I also added a checkbox to the ***/vat-return (VATReturn.jsx)*** page, to allow the user to confirm the data they entered is correct. This is required by the HMRC in the API request to submit a VAT return.

VATReturn.jsx

```
48  55                          <br />
    56  +                       <Checkbox
    57  +                               value={finalised}
    58  +                               onChange={setFinalised}
    59  +                               label="I confirm this data is correct"
    60  +                       />
    61  +                       <br />
49  62                          <Button fullWidth size="lg" color="yellow.6">
50  63                                  Submit
51  64                          </Button>
```

/vat-return



Created a table for invoice items

Invoice items are displayed in a table, each created through a popup form, this approach allows the user to see all of their added items in an easy-to-read format.

CreateUpdateInvoice.jsx

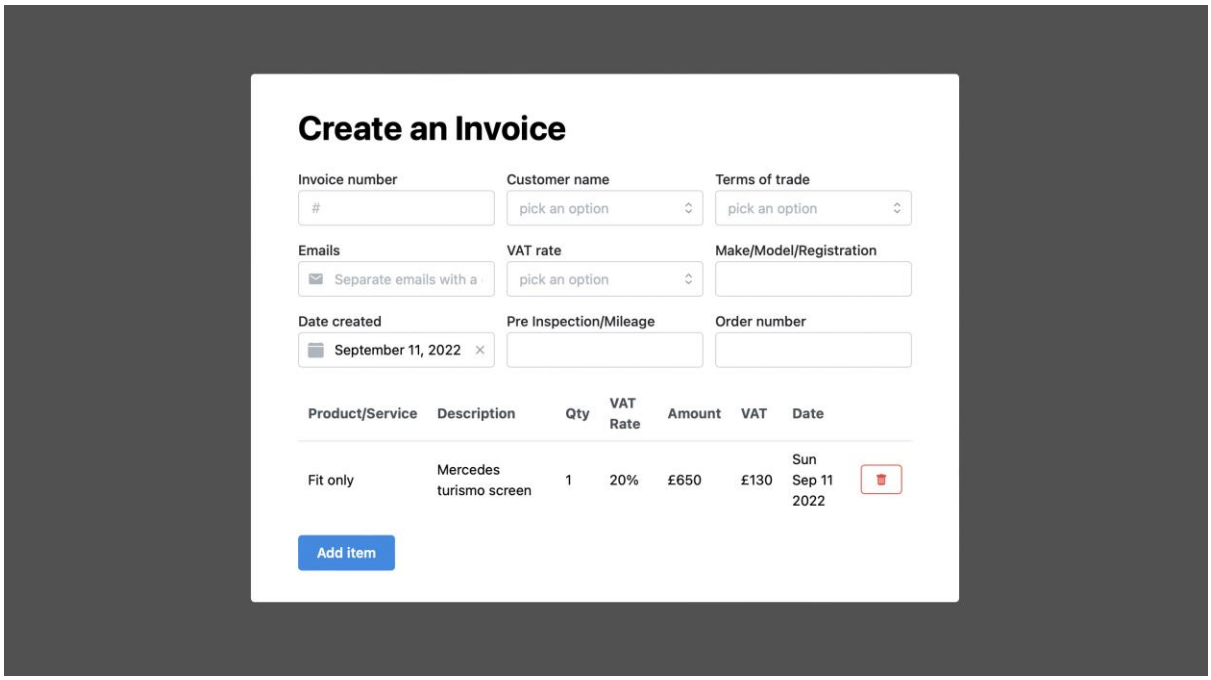
```
const testItem = {
        id: 0,
        service: "Fit only",
        description: "Mercedes turismo screen",
        quantity: 1,
        vatRate: "20%",
        price: "£650",
        vat: `£${650 * 0.2}`,
        date: new Date(),
};

const rows = invoiceItems.map((item) => (
        <tr key={item.id}>
                <td>{item.service}</td>
                <td>{item.description}</td>
                <td>{item.quantity}</td>
                <td>{item.vatRate}</td>
                <td>{item.price}</td>
                <td>{item.vat}</td>
                <td>{item.date.toDateString()}</td>
                <td>
                        <Button color="red.7" variant="outline" size="xs">
                                <BsFillTrashFill />
                        </Button>
                </td>
        </tr>
));
```

testItem is the format of how data will be stored by the backend and *rows* is what is rendered to the webpage, this is created through mapping each item into a *<tr>* (table row) element.

```
                        <Table>
                                <thead>
                                        <tr>
                                                <th>Product/Service</th>
                                                <th>Description</th>
                                                <th>Qty</th>
                                                <th>VAT Rate</th>
                                                <th>Amount</th>
                                                <th>VAT</th>
                                                <th>Date</th>
                                                <th></th>
                                        </tr>
                                </thead>
                                <tbody>{rows}</tbody>
                        </Table>
                        <br />
                        <Button
                                onClick={() => {
                                        setInvoiceItems([...invoiceItems, testItem]);
                                }}
                        >
                                Add item
                        </Button>
```

The *<Table>* tag is from the mantine UI library and just adds a format to the already existing html element. To test that items are added correctly, I added a button that appends a new testItem to the end of invoiceItems. This works but its major flaw is that the ID for each item is the same, so when it comes to adding a delete button for each item, the program won't know which one to delete.

/invoice/create



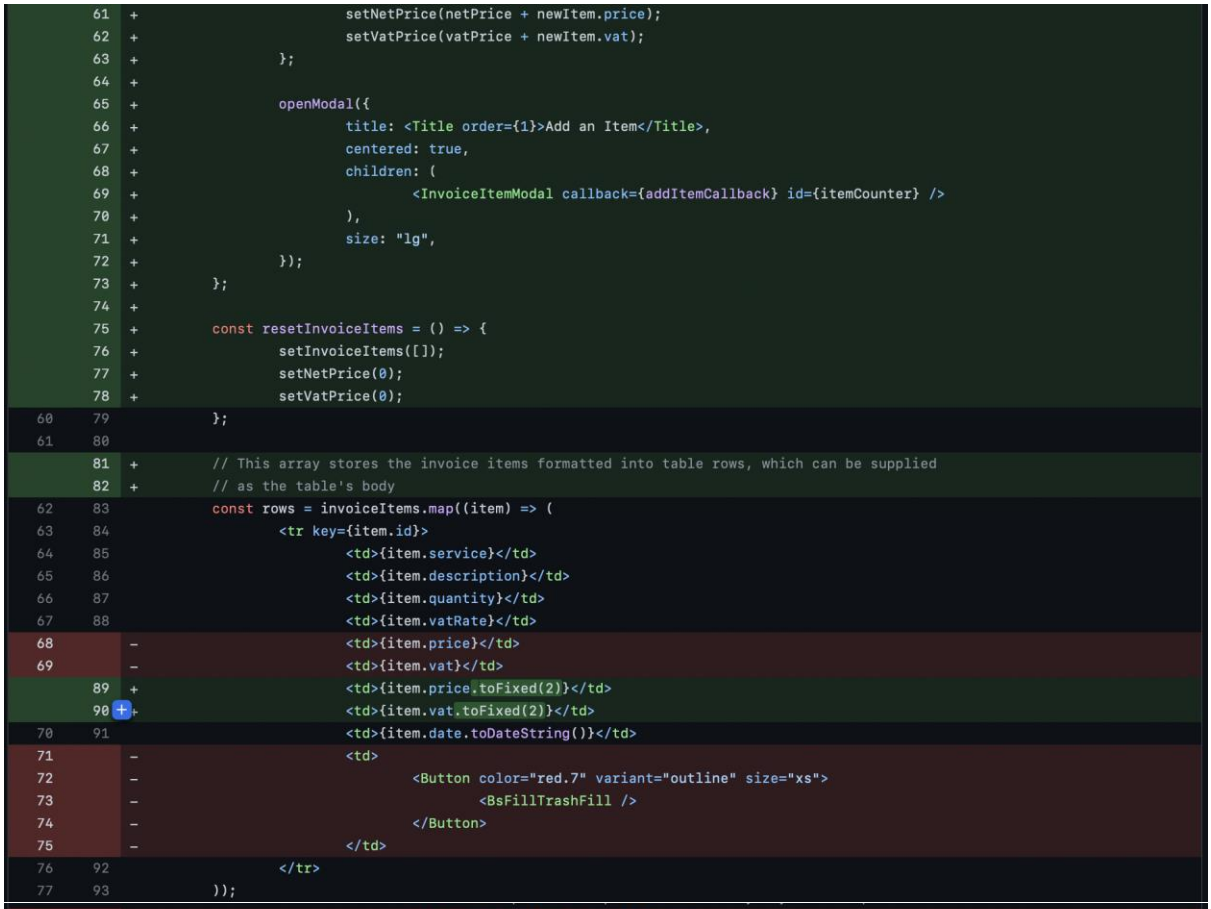
The delete button has no functionality yet, but will delete the corresponding item when implemented.

Finished the Invoice UI

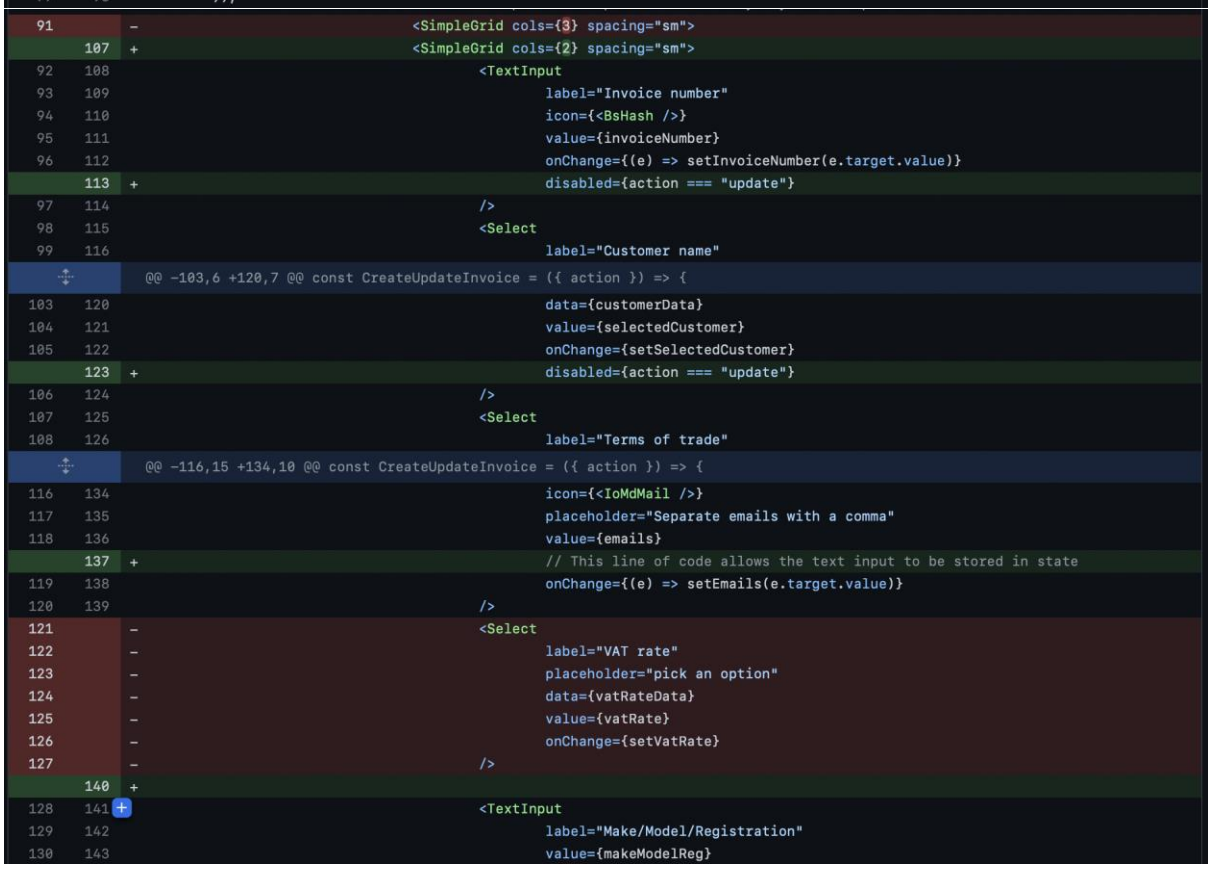
I added the rest of the UI for the create/update invoice pages, disabling fields that the user shouldn't change when updating. I also added a Net/VAT/Total calculator and submit & back buttons at the bottom of the page.

CreateUpdateInvoice.jsx

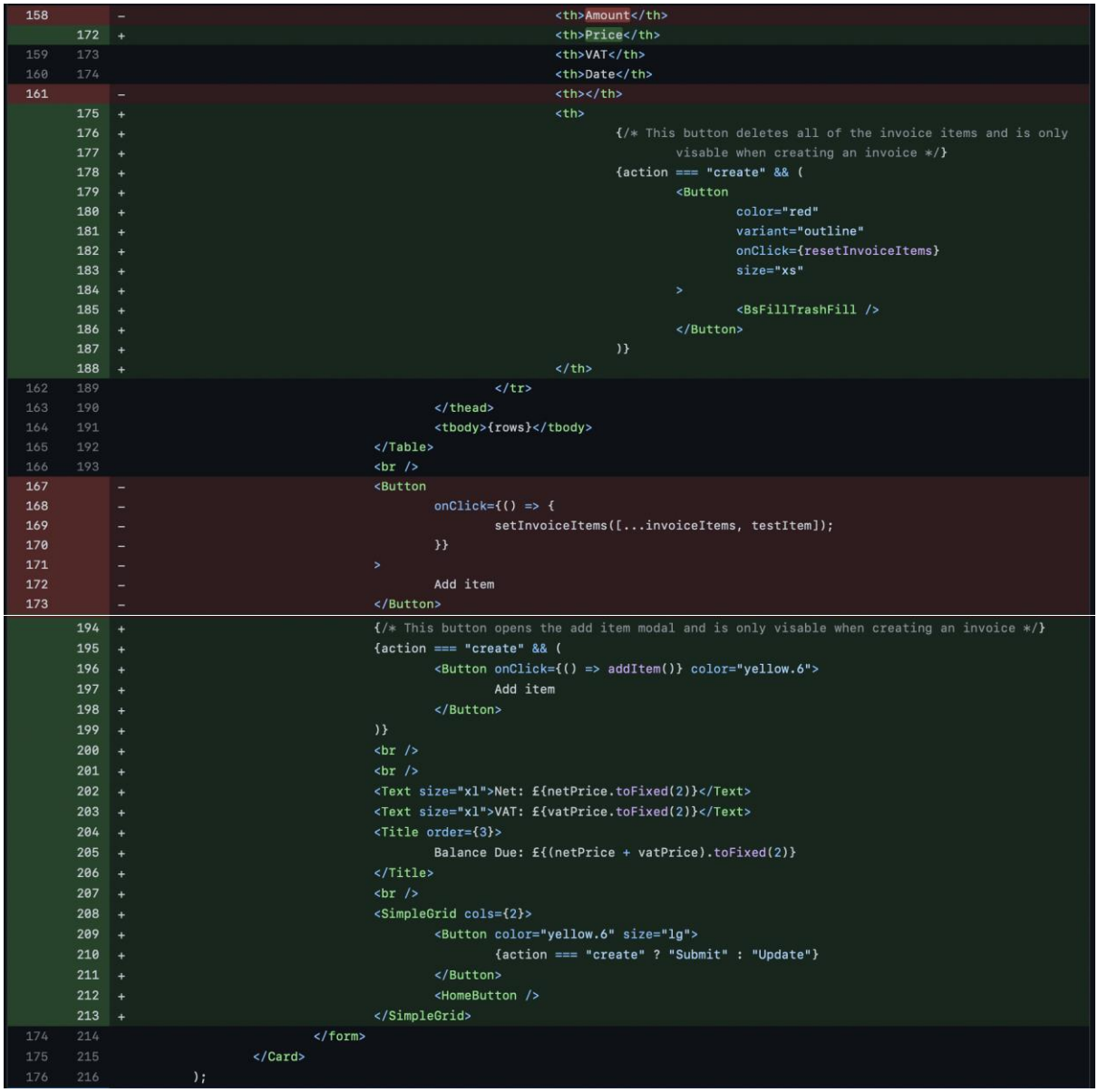
```
45  -       const [vatRate, setVatRate] = useState(null);
46  43      const [makeModelReg, setMakeModelReg] = useState("");
47  44      const [preInspection, setPreInspection] = useState("");
48  45      const [orderNumber, setOrderNumber] = useState("");
49  46      const [invoiceItems, setInvoiceItems] = useState([]);
    47  +   const [itemCounter, setItemCounter] = useState(0);
    48  +
    49  +   const [netPrice, setNetPrice] = useState(0);
    50  +   const [vatPrice, setVatPrice] = useState(0);
    51  +
    52  +   // This function is responsible for opening the add item form & adding the submitted item
    53  +   // to the list invoiceItems
    54  +   const addItem = () => {
    55  +           // This callback function allows the InvoiceItemModal component to alter the state
    56  +           // of its parent component.
    57  +           const addItemCallback = (newItem) => {
    58  +                   setInvoiceItems([...invoiceItems, newItem]);
    59  +                   setItemCounter(itemCounter + 1);
50  60
51  -       const testItem = {
52  -               id: 0,
53  -               service: "Fit only",
54  -               description: "Mercedes turismo screen",
55  -               quantity: 1,
56  -               vatRate: "20%",
57  -               price: "£650",
58  -               vat: `£${650 * 0.2}`,
59  -               date: new Date(),
```



```
+                        setNetPrice(netPrice + newItem.price);
+                        setVatPrice(vatPrice + newItem.vat);
+                };
+
+                openModal({
+                        title: <Title order={1}>Add an Item</Title>,
+                        centered: true,
+                        children: (
+                                <InvoiceItemModal callback={addItemCallback} id={itemCounter} />
+                        ),
+                        size: "lg",
+                });
+        };
+
+        const resetInvoiceItems = () => {
+                setInvoiceItems([]);
+                setNetPrice(0);
+                setVatPrice(0);
        };

+        // This array stores the invoice items formatted into table rows, which can be supplied
+        // as the table's body
        const rows = invoiceItems.map((item) => (
                <tr key={item.id}>
                        <td>{item.service}</td>
                        <td>{item.description}</td>
                        <td>{item.quantity}</td>
                        <td>{item.vatRate}</td>
-                       <td>{item.price}</td>
-                       <td>{item.vat}</td>
+                       <td>{item.price.toFixed(2)}</td>
+                       <td>{item.vat.toFixed(2)}</td>
                        <td>{item.date.toDateString()}</td>
-                       <td>
-                               <Button color="red.7" variant="outline" size="xs">
-                                       <BsFillTrashFill />
-                               </Button>
-                       </td>
                </tr>
        ));
```



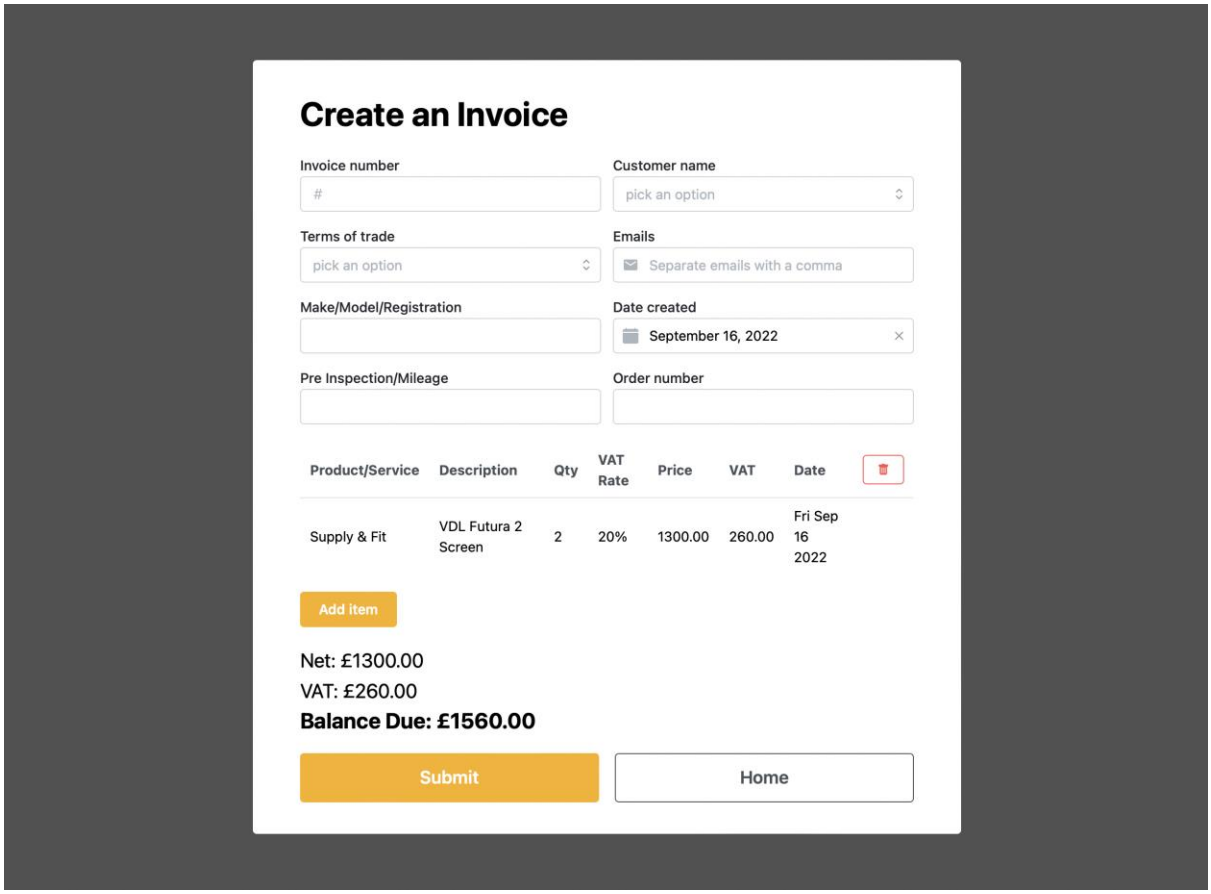
```
-                               <SimpleGrid cols={3} spacing="sm">
+                               <SimpleGrid cols={2} spacing="sm">
                                        <TextInput
                                                label="Invoice number"
                                                icon={<BsHash />}
                                                value={invoiceNumber}
                                                onChange={(e) => setInvoiceNumber(e.target.value)}
+                                               disabled={action === "update"}
                                        />
                                        <Select
                                                label="Customer name"
@@ -103,6 +120,7 @@ const CreateUpdateInvoice = ({ action }) => {
                                                data={customerData}
                                                value={selectedCustomer}
                                                onChange={setSelectedCustomer}
+                                               disabled={action === "update"}
                                        />
                                        <Select
                                                label="Terms of trade"
@@ -116,15 +134,10 @@ const CreateUpdateInvoice = ({ action }) => {
                                                icon={<IoMdMail />}
                                                placeholder="Separate emails with a comma"
                                                value={emails}
+                                               // This line of code allows the text input to be stored in state
                                                onChange={(e) => setEmails(e.target.value)}
                                        />
-                                       <Select
-                                               label="VAT rate"
-                                               placeholder="pick an option"
-                                               data={vatRateData}
-                                               value={vatRate}
-                                               onChange={setVatRate}
-                                       />
+
                                        <TextInput
                                                label="Make/Model/Registration"
                                                value={makeModelReg}
```



```
158     -                                   <th>Amount</th>
        172 +                               <th>Price</th>
159     173                                 <th>VAT</th>
160     174                                 <th>Date</th>
161     -                                   <th></th>
        175 +                               <th>
        176 +                                       {/* This button deletes all of the invoice items and is only
        177 +                                              visable when creating an invoice */}
        178 +                                       {action === "create" && (
        179 +                                               <Button
        180 +                                                       color="red"
        181 +                                                       variant="outline"
        182 +                                                       onClick={resetInvoiceItems}
        183 +                                                       size="xs"
        184 +                                               >
        185 +                                                       <BsFillTrashFill />
        186 +                                               </Button>
        187 +                                       )}
        188 +                               </th>
162     189                                         </tr>
163     190                                 </thead>
164     191                                 <tbody>{rows}</tbody>
165     192                         </Table>
166     193                         <br />
167     -                           <Button
168     -                                   onClick={() => {
169     -                                           setInvoiceItems([...invoiceItems, testItem]);
170     -                                   }}
171     -                           >
172     -                                   Add item
173     -                           </Button>
        194 +                       {/* This button opens the add item modal and is only visable when creating an invoice */}
        195 +                       {action === "create" && (
        196 +                               <Button onClick={() => addItem()} color="yellow.6">
        197 +                                       Add item
        198 +                               </Button>
        199 +                       )}
        200 +                       <br />
        201 +                       <br />
        202 +                       <Text size="xl">Net: £{netPrice.toFixed(2)}</Text>
        203 +                       <Text size="xl">VAT: £{vatPrice.toFixed(2)}</Text>
        204 +                       <Title order={3}>
        205 +                               Balance Due: £{(netPrice + vatPrice).toFixed(2)}
        206 +                       </Title>
        207 +                       <br />
        208 +                       <SimpleGrid cols={2}>
        209 +                               <Button color="yellow.6" size="lg">
        210 +                                       {action === "create" ? "Submit" : "Update"}
        211 +                               </Button>
        212 +                               <HomeButton />
        213 +                       </SimpleGrid>
174     214                 </form>
175     215             </Card>
176     216         );
```

I removed the VAT rate selecion from the main form as it was redundant. I added individual ones for each item on the invoice. I also removed the individual delete buttons as there was no feasable way I could get it to function. Sometimes it worked, other times it deleted random objects. So instead, I added a delete all button to allow the user to start over.

/invoice/create



Next, I created a form that allows users to add items to invoices. I had this form use MantineUI's modals manager.

InvoiceItemModal.jsx

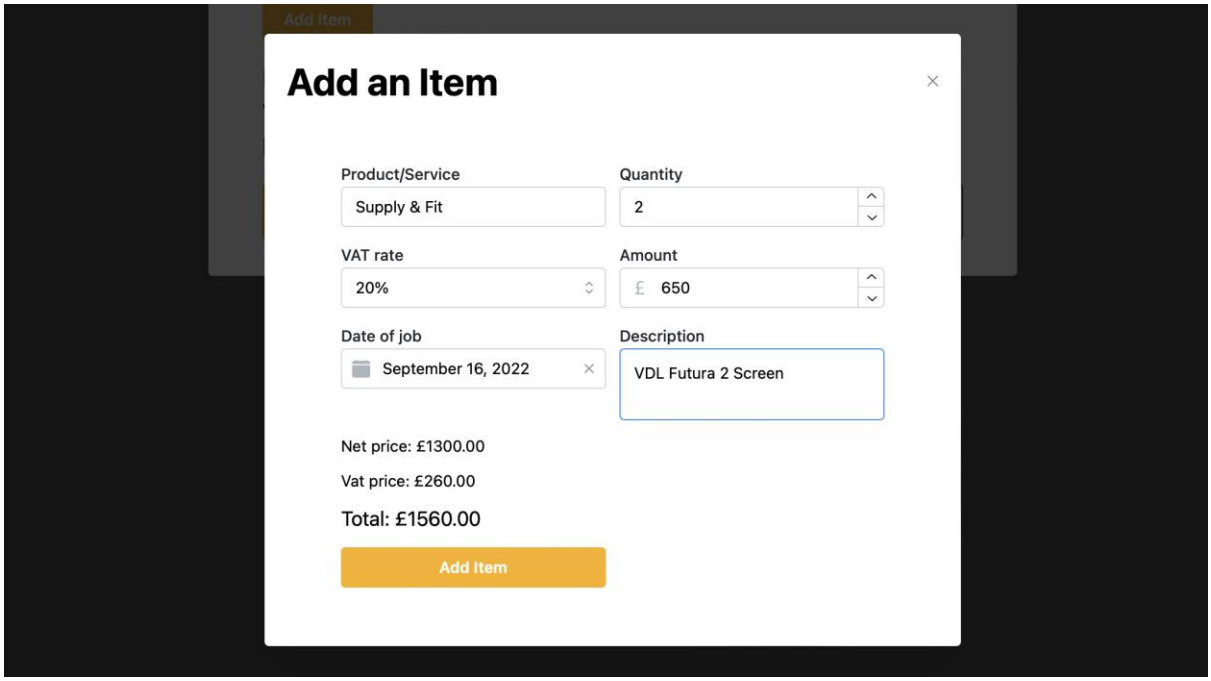
115 lines (108 sloc) | 2.76 KB Raw Blame

```jsx
import {
        Button,
        NumberInput,
        Select,
        SimpleGrid,
        Text,
        Textarea,
        TextInput,
} from "@mantine/core";
import { DatePicker } from "@mantine/dates";
import { closeAllModals } from "@mantine/modals";
import { useState } from "react";
import { BsCurrencyPound, BsFillCalendarFill } from "react-icons/bs";

const InvoiceItemModal = ({ callback, id }) => {
        const [service, setService] = useState("");
        const [description, setDescription] = useState("");
        const [quantity, setQuantity] = useState(1);
        const [vatRate, setVatRate] = useState(null);
        const [amount, setAmount] = useState(0);
        const [dateCreated, setDateCreated] = useState(new Date());

        const vatRateData = [
                { label: "20%", value: 0.2 },
                { label: "Exempt", value: "NA" },
                { label: "Zero", value: 0 },
        ];

        const handleButtonPressed = () => {
                let formattedVatRate;
                switch (vatRate) {
                        case 0.2:
                                formattedVatRate = "20%";
                                break;
                        case 0:
                                formattedVatRate = "Zero";
                                break;
                        default:
                                formattedVatRate = "Exempt";
                                break;
                }

                callback({
                        id,
                        service,
                        description,
                        quantity,
                        vatRate: formattedVatRate,
                        price: netPrice,
                        vat: vatPrice,
                        date: dateCreated,
                });
                closeAllModals();
        };

        // These variables update in real time when the state of the form is changed
        let netPrice = amount * quantity;
        let vatPrice = netPrice * (vatRate === "NA" ? 0 : vatRate);
        let total = netPrice + vatPrice;

        return (
                <div className="card">
                        <form>
                                <SimpleGrid cols={2} spacing="sm">
                                        <TextInput
                                                label="Product/Service"
                                                value={service}
                                                onChange={(e) => setService(e.target.value)}
                                        />
                                        <NumberInput
                                                label="Quantity"
                                                value={quantity}
                                                onChange={(e) => setQuantity(e)}
                                        />
                                        <Select
                                                label="VAT rate"
                                                placeholder="pick an option"
                                                data={vatRateData}
                                                value={vatRate}
                                                onChange={setVatRate}
                                        />
```

```
                                <NumberInput
                                        label="Amount"
                                        icon={<BsCurrencyPound />}
                                        value={amount}
                                        onChange={(e) => setAmount(e)}
                                />
                                <DatePicker
                                        label="Date of job"
                                        icon={<BsFillCalendarFill />}
                                        value={dateCreated}
                                        onChange={setDateCreated}
                                />
                                <Textarea
                                        label="Description"
                                        value={description}
                                        onChange={(e) => setDescription(e.target.value)}
                                />
                                <SimpleGrid cols={1} spacing="xs">
                                        <Text size="sm">Net price: £{netPrice.toFixed(2)}</Text>
                                        <Text size="sm">Vat price: £{vatPrice.toFixed(2)}</Text>
                                        <Text size="lg">Total: £{total.toFixed(2)}</Text>
                                </SimpleGrid>
                                <br />
                                {/* This button submits the form and adds the item to the table in CreateUpdateInvoice */}
                                <Button onClick={handleButtonPressed} color="yellow.6">
                                        Add Item
                                </Button>
                        </SimpleGrid>
                </form>
        </div>
    );
};

export default InvoiceItemModal;
```

The ***handleButtonPressed*** function formats the VAT rate into a user-readable form rather than the one optimised for calculating the VAT total. It then calls the callback function I made in ***CreateUpdateUserInvoice.jsx*** file.

/invoice/create Add Item Modal



I also fixed a typo in ***App.jsx*** in the routes. I discovered this by trying to access the ***/invoice/update*** endpoint and getting a 404 error (page does not exist)

App.jsx

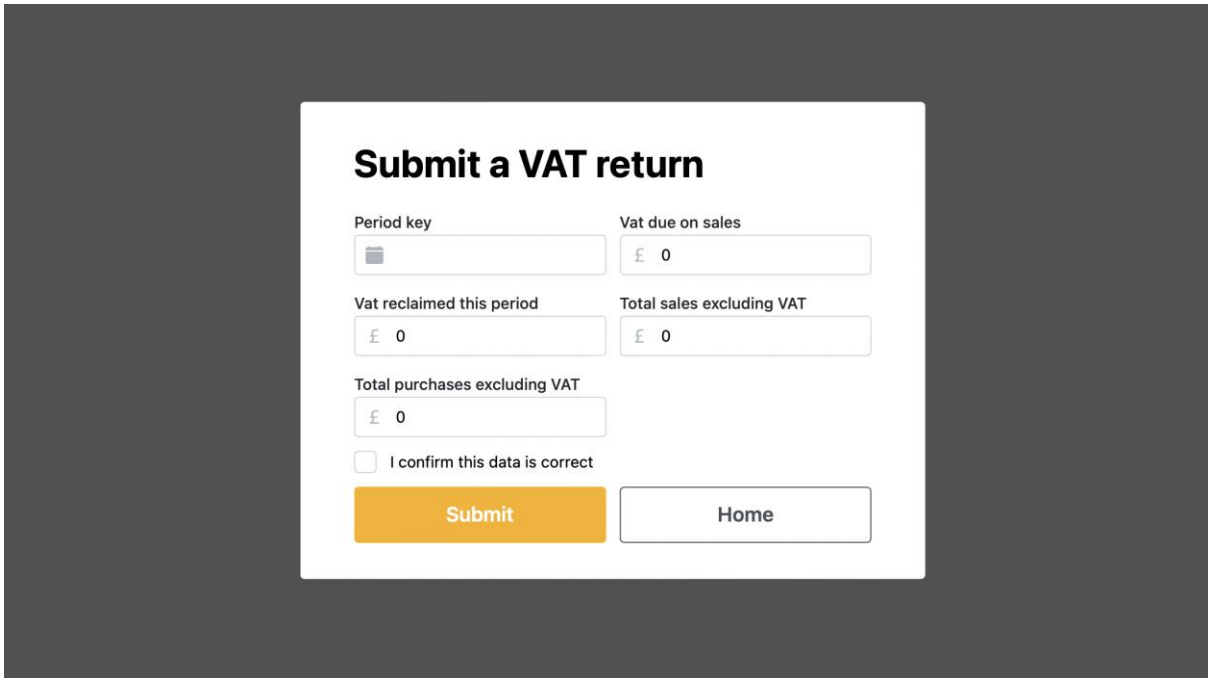
```
-                                                   element={<CreateUpdateInvoice action="/update" />}
+                                                   element={<CreateUpdateInvoice action="update" />} // Typo Fixed
```

Review

After a review meeting with my stakeholders, they suggested that I tweak the layout for the VAT return form to make it 2 columns wide rather than one (see here)

```
25                                    <SimpleGrid cols={2} spacing="sm">
```

/vat-return



I put the home button component in its own file to increase its reusablilty. I also gave it a size setting to plan for cases where a different sized button may be needed.

```jsx
import { Button } from "@mantine/core";
// This component has the sole purpose of sending the user back to the home page,
// regardless of where they are on the site.

const HomeButton = ({ size }) => {
        // clickable functionality will be added in prototype 3
        const handleClick = () => {};
        return (
                <Button
                        fullWidth
                        // If a size isn't supplied use large as a default
                        size={size || "lg"}
                        color="gray.7"
                        variant="outline"
                        onClick={handleClick}
                >
                        Home
                </Button>
        );
};

export default HomeButton;
```

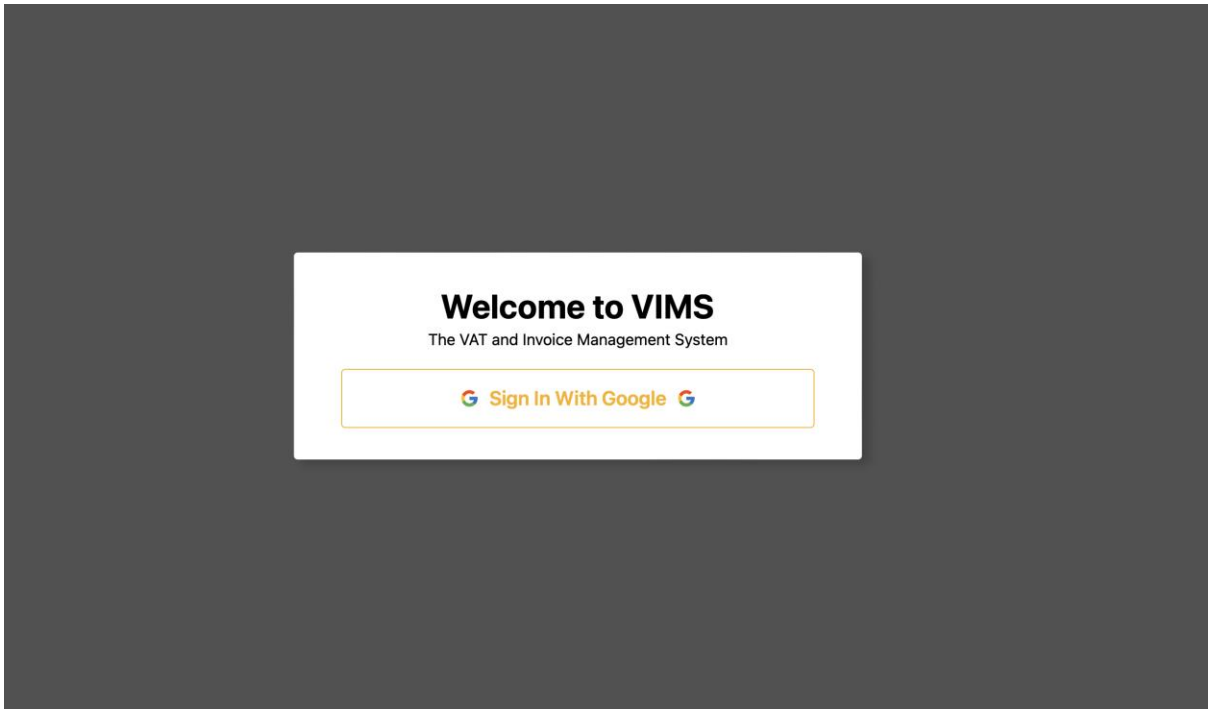
Prototype 3

https://github.com/daniel-555/A2-CS-Project/tree/Prototype3

In prototype 3, I added the login system. This consists of three parts.

SignedOut.jsx

```
// UI-related
import { Button, Card, Text, Title } from "@mantine/core";
import { FcGoogle } from "react-icons/fc";

// Firebase (backend)
import { GoogleAuthProvider, signInWithPopup } from "firebase/auth";
import { auth } from "../../firebase/firebase-init";

const SignedOut = () => {
        const signInWithGoogle = () => {
                const provider = new GoogleAuthProvider();
                signInWithPopup(auth, provider);
        };

        return (
                <Card className="card center" sx={{ width: "40%", textAlign: "center" }}>
                        <Title order={1}>Welcome to VIMS</Title>
                        <Text>The VAT and Invoice Management System</Text>
                        <br />
                        <Button
                                size="xl"
                                variant="outline"
                                color="yellow.6"
                                leftIcon={<FcGoogle />}
                                rightIcon={<FcGoogle />}
                                onClick={signInWithGoogle}
                                fullWidth
                        >
                                Sign In With Google
                        </Button>
                </Card>
        );
};

export default SignedOut;
```



This renders the page that is displayed to a user when they are signed out. Firebase detects whether a user is signed in on a specific session using the pre-built ***useAuthState*** hook. This returns false if a user is signed out, and an object containing the user's data (uid, username, email, etc.) if they are signed in. By using a ternary operator in the root ***App*** component, the page can decide which content to render.

```
{user ? <SignedIn /> : <SignedOut />}
```

SignedIn.jsx

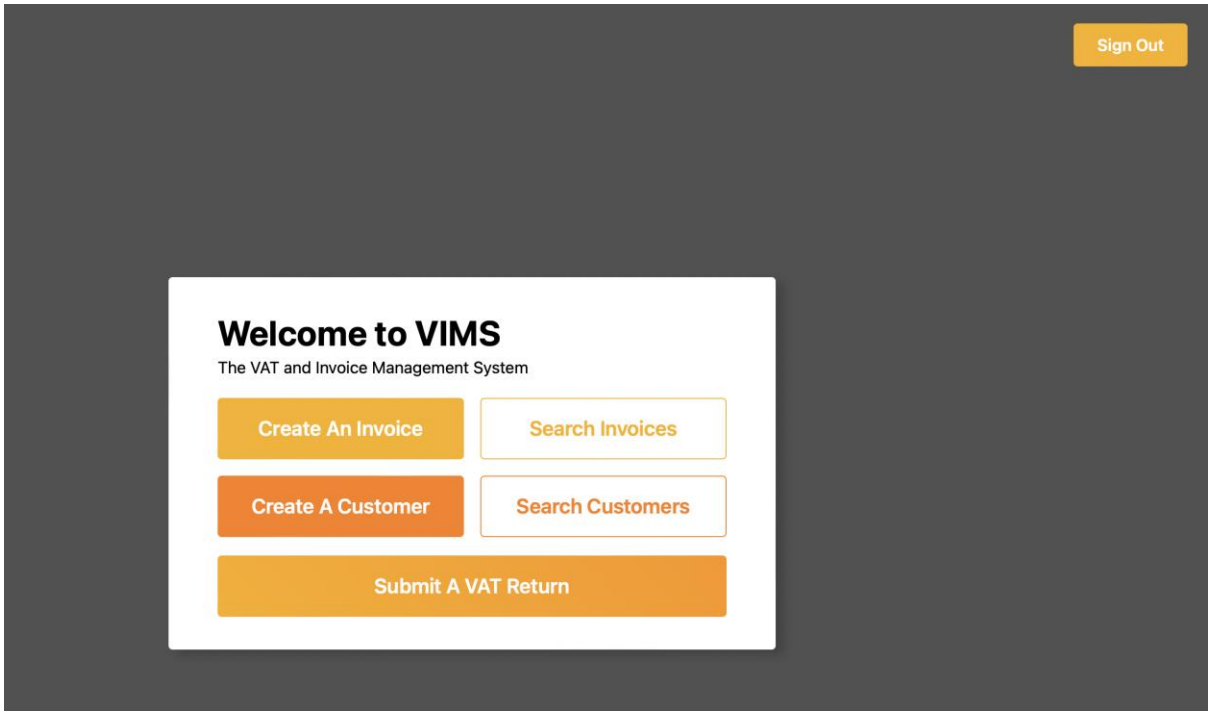
```
const SignedIn = () => {
        return (
                <>
                        <SignOutButton />
                        <BrowserRouter>
                                {/* Directory of all routes on the site */}
                                <Routes>
                                        <Route path="/" element={<HomePage />} />
                                        <Route path="/vat-return" element={<VATReturn />} />
                                        {/* Routes can be nested for organisation as shown below */}
                                        <Route path="/customer">
                                                <Route
                                                        path="create"
                                                        element={<CreateUpdateCustomer action="create" />}
                                                />
                                                <Route
                                                        path="update/:customerID"
                                                        element={<CreateUpdateCustomer action="update" />}
                                                />
                                        </Route>
                                        <Route path="/invoice">
                                                <Route
                                                        path="create"
                                                        element={<CreateUpdateInvoice action="create" />}
                                                />
                                                <Route
                                                        // the :invoice parameter is a user input in the url
                                                        path="update/:invoice"
                                                        element={<CreateUpdateInvoice action="update" />}
                                                />
                                                <Route path="view/:invoice" element={<ViewInvoice />} />
                                        </Route>
                                        <Route path="/database/:table" element={<DatabaseQuery />} />
                                </Routes>
                        </BrowserRouter>
                </>
        );
};

export default SignedIn;
```

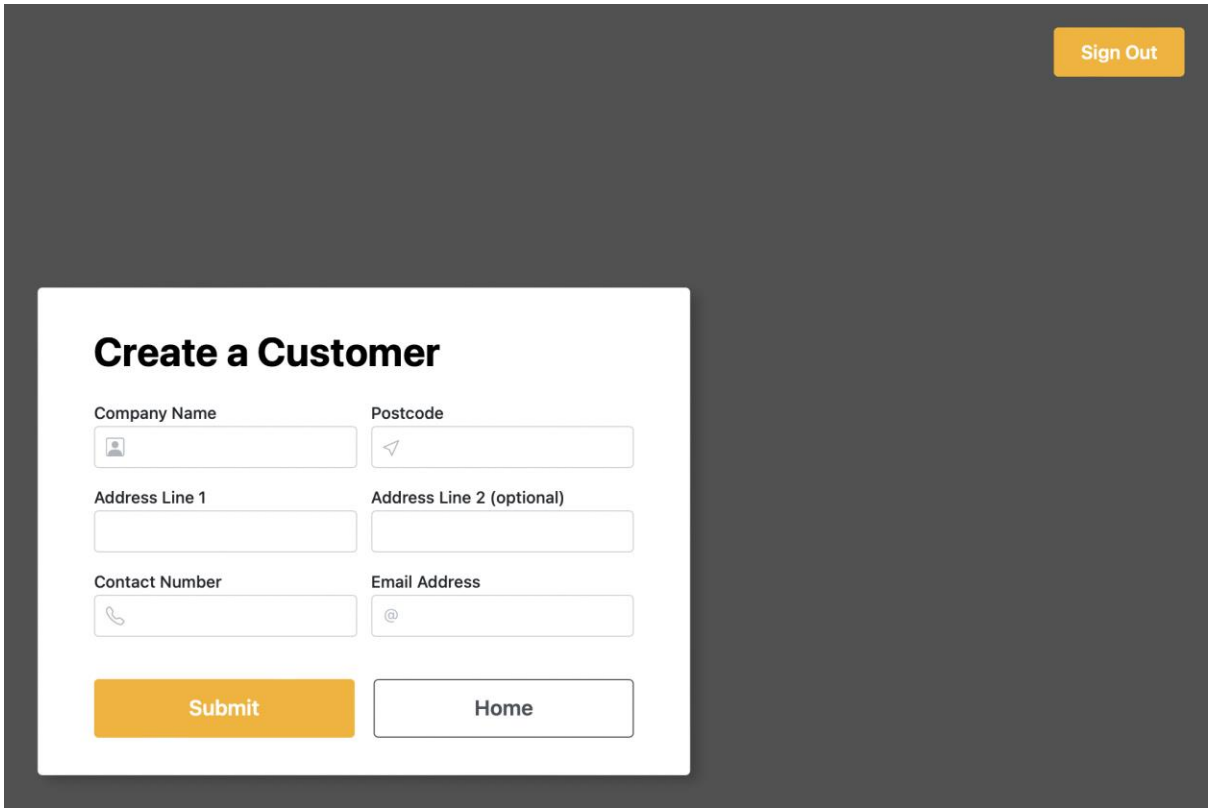
This code is from ***Prototype 2*** and directs the user to each page on the site. The only tweak from then is the added ***SignOutButton*** component.

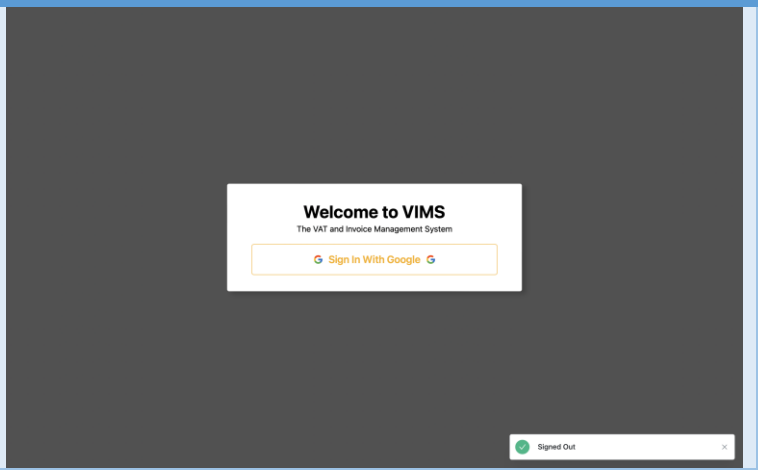
And ***SignOutButton.jsx***

```
const SignOutButton = () => {
        // This activates when the button is pressed
        const logout = () => {
                // Firebase's built-in funciton that signs a user out
                signOut(auth)
                        // This is JavaScript's promise syntax. The code in .then isn't
                        // executed until the user has been confirmed signed out
                        .then(() =>
                                // This code displays a notification on the screen
                                showNotification({
                                        title: "Signed Out",
                                        icon: <AiOutlineCheck />,
                                        color: "teal",
                                        autoClose: 2000,
                                })
                        )
                        .catch((error) => console.log(error));
        };

        return (
                <Affix position={{ top: 20, right: 20 }}>
                        <Button color="yellow.6" size="md" onClick={logout}>
                                Sign Out
                        </Button>
                </Affix>
        );
};
```

this button is affixed to the top right of every page in the ***SignedIn*** component (examples below)





#	Test	Pass / Fail	Evidence
1	When the button is pressed, the user is signed out and sees the SignedOut page. A notification is also displayed.	PASS	

I refactored the state in ***CreateUpdateCustomer.jsx*** to be a javascript object rather than individual variables as it makes code faster to package and send off to the backend. This meant I had to change the way that state was edited, I did this by making a general function. I didn't plan for this function in the design stage.

```
41          // Generalised function to change state in the form.
42          // maxLength input is to limit the length of strings that shouldn't be long
43          // the default maximum is 10 characters
44          const changeState = (value, key, maxLength = 10) => {
45                  if (value.length <= maxLength) {
46                          setCustomerData((customerData) => ({ ...customerData, [key]: value }));
47                  }
48          };
```

Line 46 is where the data is added to state, by using javascript's spread operator I can update a specific key/value pair whilst leaving the rest of the object in the current state untouched. This function is called by each component using their onChange event listener, for example:

```
<TextInput
        label="Company Name"
        value={customerData.companyName}
        icon={<BsPersonSquare />}
        onChange={(e) =>
                changeState(
                        e.target.value,
                        "companyName",
                        fieldMaxLengths.companyName
                )
        }
/>
```

The default state for the form is now generated as shown below

```
const customerDefault = {
        companyName: "",
        address1: "",
        address2: "",
        postcode: "",
        contactNo: "",
        email: "",
};

const [customerData, setCustomerData] = useState(customerDefault);
```

To test the function I added a console.log() line after the code of the function

```
const changeState = (value, key, maxLength = 10) => {
    if (value.length <= maxLength) {
        setCustomerData((customerData) => ({ ...customerData, [key]: value }));
        console.log(`Successfully updated ${key} field`);
    }
};
```

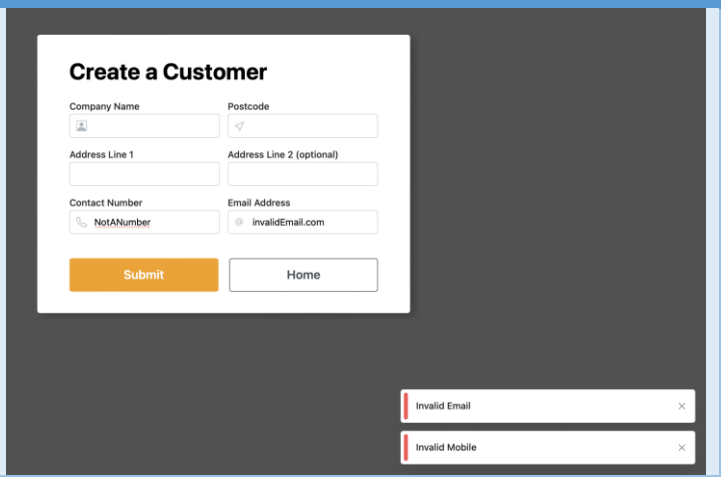
Test	Pass / Fail	Evidence
Edit text in each input and await the success log	PASS	Successfully updated companyName field Successfully updated postcode field Successfully updated address1 field 3 Successfully updated address2 field Successfully updated contactNo field Successfully updated email field

Also in ***CreateUpdateCustomer.jsx***, is the function that handles when the submit button is pressed.

```
50          // Runs when the submit button is pressed
51          const handleSubmit = () => {
52                  // validate that fields are in the correct format
53                  let formOk = true;
54
55                  if (validateEmail(customerData.email) === false) {
56                          showNotification({ title: "Invalid Email", color: "red" });
57                          formOk = false;
58                  }
59                  if (validateMobile(customerData.contactNo) === false) {
60                          showNotification({ title: "Invalid Mobile", color: "red" });
61                          formOk = false;
62                  }
63
64                  // Break from the function if an error was detected above
65                  if (formOk === false) return;
66
67                  if (action === "create") {
68                          createCustomer(customerData);
69
70                          // Clear all the forms so data cannot be double submitted
71                          setCustomerData(customerDefault);
72                  }
73                  if (action === "update") {
74                          updateCustomer(customerID, customerData);
75                  }
76          };
```

validateEmail and ***validateMobile*** are both validation functions that are stored in a different file (see here).

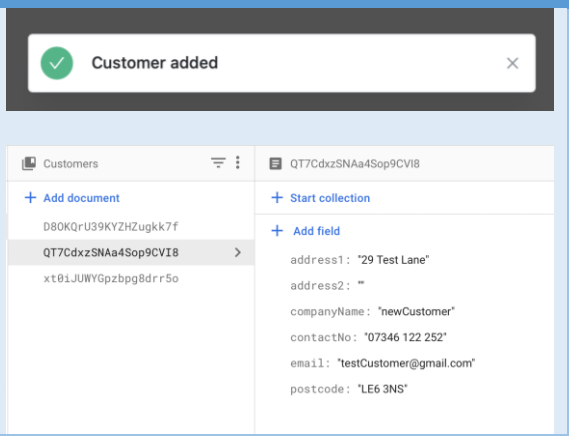
If validation fails, a notification is displayed telling the user what is wrong and the function is exited before any interaction is done with the database. The ***showNotification*** function is a function that comes with the MantineUI library.

Test	Pass / Fail	Evidence
Input invalid data to make sure that errors in emails and mobiles are caught	PASS	

If the user is on the create page, the ***createCustomer*** function shown below is ran. If the user is on the update page, the ***updateCustomer*** function is ran.

```
// UI-related
import { AiOutlineCheck } from "react-icons/ai";
import { showNotification, updateNotification } from "@mantine/notifications";

// Firebase (backend)
import { db } from "../../firebase/firebase-init";
import { addDoc, collection } from "firebase/firestore";

const createCustomer = (customerData) => {
        // Tell the user that the action has started
        showNotification({
                id: "await-add",
                title: "Form submitted",
                message: "Adding customer to database",
                loading: true,
                autoClose: false,
                disallowClose: true,
        });

        // reference to the Customers collection
        const collectionRef = collection(db, "Customers");

        // Add the customerData to the database
        addDoc(collectionRef, customerData).then(() => {
                // Tell the user that the action has been completed
                updateNotification({
                        id: "await-add",
                        title: "Customer added",
                        icon: <AiOutlineCheck />,
                        color: "teal",
                });

                // Wait 1 second then refresh the page
                setTimeout(() => window.location.reload(), 1000);
        });
};

export default createCustomer;
```

This is the function that interacts with the database and creates a new customer document with the inputted data. The customer ID is generated automatically by firebase. No validation is needed as data inputted has already been sanitized by the ***handleSubmit*** function above.

#	Test	Pass / Fail	Evidence
8	Inputted data will be added to the database and show in firebase's database UI.	PASS	

For validation across my project I am using custom functions that return true or false based on whether the inputted data is of the correct format.

validateEmail

```
1  export const validateEmail = (email) => {
2          // regex is a way of generalising a formt of string
3          const validEmail =
4                  //     any set of characters      @ domain     .  com, co.uk, etc
5                  /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*$/;
6
7          if (email.match(validEmail)) {
8                  // the email is of the correct format
9                  return true;
10         } else {
11                 // email is invalid
12                 return false;
13         }
14 };
```

This function is for email validation. The complicated regular expression (regex) on line 5 is used to check if emails are in the correct format.

validateMobile

```
16  export const validateMobile = (number) => {
17          // Check that the given input only contains numbers & spaces
18          const validMobile = /^[0-9 ]{1,}$/;
19
20          if (number.match(validMobile)) {
21                  return true;
22          } else {
23                  return false;
24          }
25  };
```

This function checks that mobile numbers only consist of numbers and spaces – for easy reading. It also checks that the input has at least one number (shown in red).

validateInvoiceNumber

```
27  export const validateInvoiceNumber = (invoiceNumber) => {
28          // Check that invoice number is INV followed by 4 digits
29          const validNumber = /^INV[0-9]{4}$/;
30
31          if (invoiceNumber.match(validNumber)) {
32                  return true;
33          } else {
34                  return false;
35          }
36  };
```

Invoices are automatically generated by the ***generateInvoiceNumber*** function, but validation has been added for the invoice number in case my stakeholders want to create their own in the future.

#	Test	Pass / Fail	Evidence

2	validateEmail	PASS	`console.log(validateEmail("test@email.com")) // True`
			`console.log(validateEmail("customerEmail.com")) // False`
			`console.log(validateEmail("testEmail@test-domain.co.uk")) // True`
			`console.log(validateEmail("EMAIL")) // False`
			`true`
			`false`
			`true`
			`false`

3	validateMobile	PASS	`console.log(validateMobile("01234 212 324")) // True`
			`console.log(validateMobile("  ")) // False`
			`console.log(validateMobile("07543123456")) // True`
			`console.log(validateMobile("MOBILE")) // False`
			`true`
			`false`
			`true`
			`false`

4	validateInvoiceNumber	PASS	`console.log(validateInvoiceNumber("INV0222")) // True`
			`console.log(validateInvoiceNumber("INV012")) // False`
			`console.log(validateInvoiceNumber("INV1235")) // True`
			`console.log(validateInvoiceNumber("INVOICE")) // False`
			`true`
			`false`
			`true`
			`false`

When the desired page is ***/customer/update/{customerID}***, some data needs to be fetched when the page is first loaded. I do this with the useEffect react hook. Data is fetched using the ***{customerID}*** by the ***getCustomerInfo*** function.

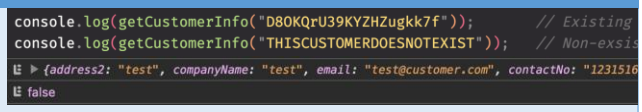
```
// Runs when the page is "mounted" (ie when it is first loaded)
useEffect(() => {
        const fetchData = async () => {
                const fetchedData = await getCustomerInfo(customerID);
                if (fetchedData) setCustomerData(fetchedData);
                else notFoundNotification("customer");
        };

        // Only get customer data when on the update page
        if (action === "update") fetchData();
}, []);
```

notFoundNotification is a custom function that shows the user a notification saying “This xxxx does not exist”

getCustomerInfo

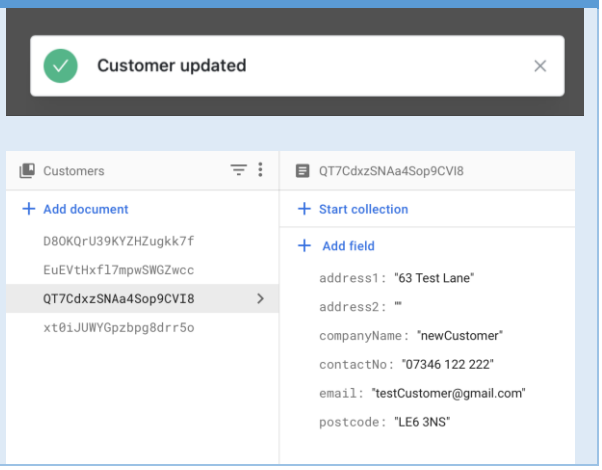
```
// retrieves the customer's info from the database and sends it back to the CreateUpdateInvoice component
const getCustomerInfo = async (customerID) => {
        const docRef = doc(db, "Customers", customerID);

        const docSnap = await getDoc(docRef);

        if (docSnap.exists()) {
                // customer has been found
                return docSnap.data();
        } else {
                // provided customerID couldn't be located
                return false;
        }
};

export default getCustomerInfo;
```

This is where customer data is fetched from the database. This function returns false if the customer can’t be located. This tells the user when the customer they’re searching for exists.

#	Test	Pass / Fail	Evidence
5	Put an existing and non-existing Customer into the getCustomerInfo function and return the result	PASS	

This function runs when the submit button is pressed on the ***/customer/update/{customerID}*** page in the ***handleSubmit*** function.

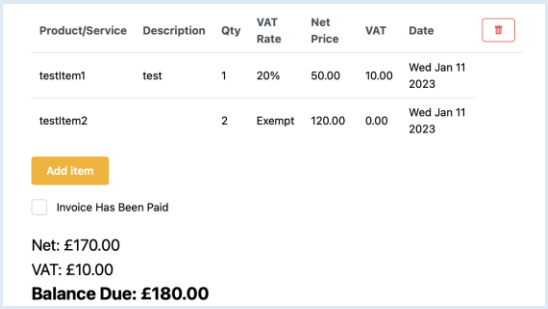
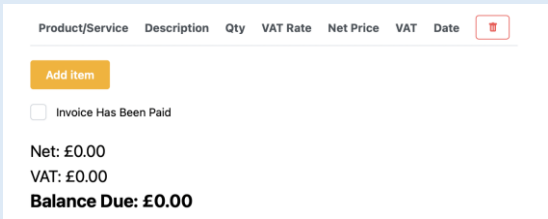
```
const updateCustomer = (customerID, customerData) => {
        const docRef = doc(db, "Customers", customerID);

        // Tell the user that the action has started
        showNotification({
                id: "await-update",
                title: "Form submitted",
                message: "Updating customer info",
                loading: true,
                autoClose: false,
                disallowClose: true,
        });

        // Update the customer in the database then tell the user that the action has completed
        setDoc(docRef, customerData).then(() => {
                updateNotification({
                        id: "await-update",
                        title: "Customer updated",
                        icon: <AiOutlineCheck />,
                        color: "teal",
                });

                // Wait 1 second then refresh the page
                setTimeout(() => window.location.reload(), 1000);
        });
};
```

#	Test	Pass / Fail	Evidence
7	Update an existing customer's data and press the submit button. A notification will be displayed, and the backend database will be updated	PASS	

For invoices I tried to get the individual delete buttons working. But every time I pressed the button, It deleted all of the invoice items or a few of them, rather than the specific one I wanted to delete. So instead of this, I added one delete button that resets the invoice item table and deletes all items.

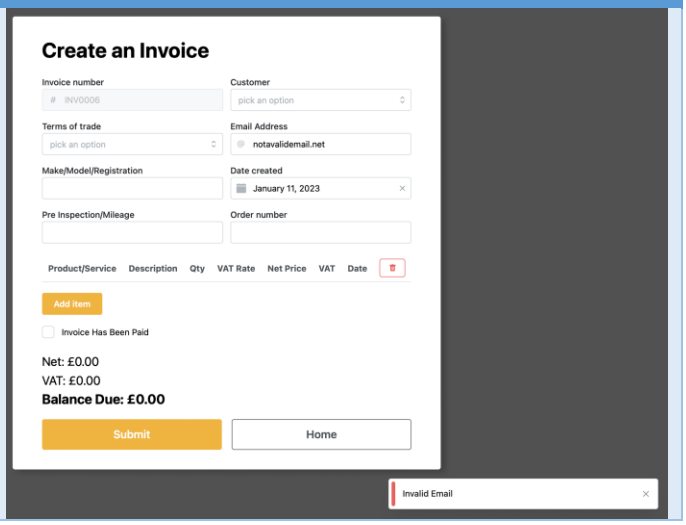
```
// Clear the invoice items table
const resetInvoiceItems = () => {
        setInvoiceItems([]);
        setNetPrice(0);
        setVatPrice(0);
};
```

This function clears the net/vat/total and invoice items table when the user presses the delete button

Test	Pass / Fail	Evidence
When the delete button is pressed the net/vat/total and invoice items would reset to zero	PASS	Before:  After: 

When the invoice is submitted, the ***handleSubmit*** function below is ran.

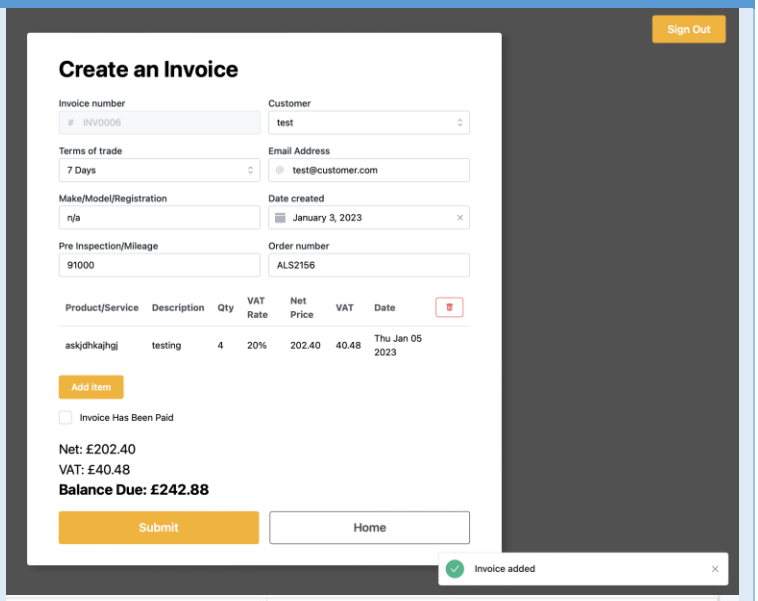
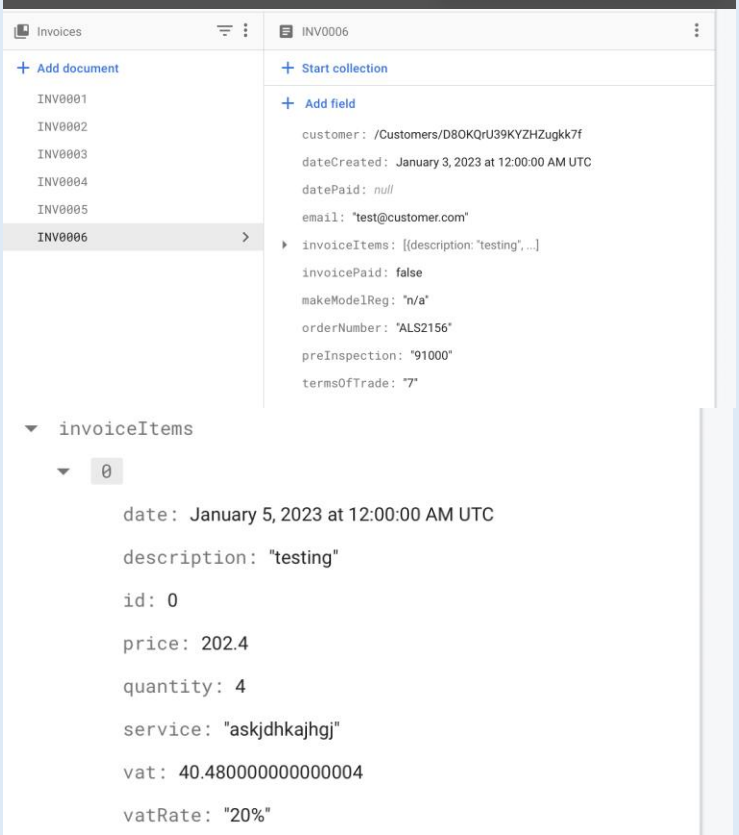
```
130         // Executes when submit button is pressed
131         const handleSubmit = () => {
132                 // Validate that all fields are in the required format
133                 let formOk = true;
134
135                 if (validateInvoiceNumber(invoiceNumber) === false) {
136                         formOk = false;
137                         showNotification({ title: "Invalid Invoice Number", color: "red" });
138                 }
139                 if (!validateEmail(email)) {
140                         formOk = false;
141                         showNotification({ title: "Invalid Email", color: "red" });
142                 }
143                 // Make sure the invoice actually contains items
144                 if (action === "create" && itemCounter === 0) {
145                         formOk = false;
146                         showNotification({
147                                 title: "You must add at least one invoice item",
148                                 color: "red",
149                         });
150                 }
151
152                 if (formOk === false) return;
153
154                 const invoiceData = {
155                         // Database reference to customer
156                         customer: doc(db, "Customers", customer),
157                         termsOfTrade,
158                         dateCreated,
159                         email,
160                         makeModelReg,
161                         preInspection,
162                         orderNumber,
163                         invoiceItems,
164                         invoicePaid,
165                         datePaid: invoicePaid ? datePaid : null,
166                 };
167
168                 if (action === "create") {
169                         createInvoice(invoiceNumber, invoiceData);
170                 } else if (action === "update") {
171                         updateInvoice(invoiceNumber, invoiceData);
172                 }
173         };
```

Test	Pass / Fail	Evidence
Input invalid data to make sure that errors in emails are caught	PASS	

If the user is on the create page, the ***createInvoice*** function is ran. If the user is on the update page, the ***updateInvoice*** function is ran.

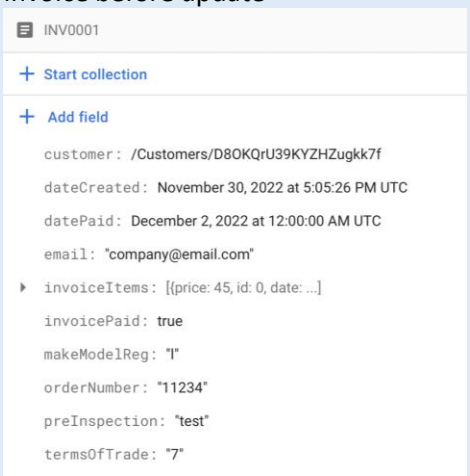
```
9   const createInvoice = (invoiceNumber, invoiceData) => {
10          // Alert the user that the function has started running
11          showNotification({
12                  id: "await-add",
13                  title: "Form submitted",
14                  message: "Adding invoice to database",
15                  loading: true,
16                  autoClose: false,
17                  disallowClose: true,
18          });
19
20          const docRef = doc(db, "Invoices", invoiceNumber);
21
22          // Create the document in the database
23          setDoc(docRef, invoiceData).then(() => {
24                  // Tell the user that the action has been completed
25                  updateNotification({
26                          id: "await-add",
27                          title: "Invoice added",
28                          icon: <AiOutlineCheck />,
29                          color: "teal",
30                  });
31                  // Wait 1 second then refresh the page
32                  setTimeout(() => window.location.reload(), 1000);
33          });
34  };
35
36  export default createInvoice;
```

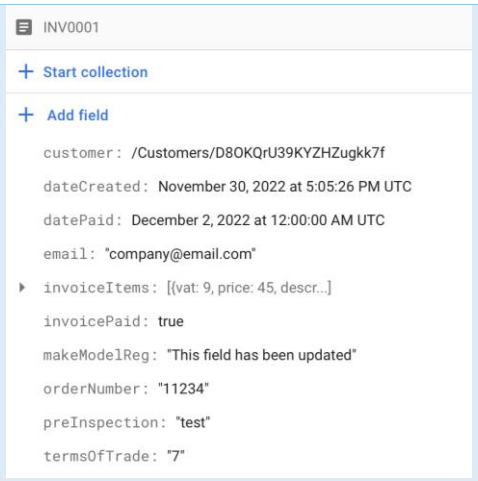
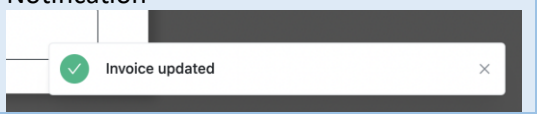
No validation is needed as data has already been checked over by the ***handleSubmit*** function

#	Test	Pass / Fail	Evidence
10	Input valid data to the function and expect a notification returned and an invoice added to the database	PASS	 

When the user is on the update page, the ***handleSubmit*** function runs the ***updateInvoice*** function below.

```
const updateInvoice = (invoiceNumber, invoiceData) => {
        const docRef = doc(db, "Invoices", invoiceNumber);
        showNotification({
                id: "await-update",
                title: "Form submitted",
                message: "Updating invoice info",
                loading: true,
                autoClose: false,
                disallowClose: true,
        });

        // Update the document in the database
        setDoc(docRef, invoiceData).then(() => {
                // Tell the user the action has been completed
                updateNotification({
                        id: "await-update",
                        title: "Invoice updated",
                        icon: <AiOutlineCheck />,
                        color: "teal",
                });
                // Wait 1 second then refresh the page
                setTimeout(() => window.location.reload(), 1000);
        });
};
```

#	Test	Pass / Fail	Evidence
9	When the function is called, the inputted invoice will be updated with the new data and a notification will be displayed	PASS	Invoice before update  Invoice after update

			 Notification 

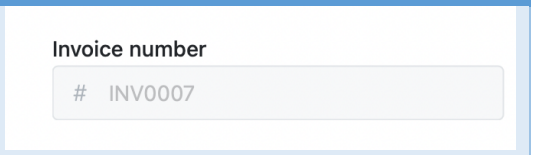
The invoice number is generated automatically by the ***generateInvoiceNumber*** function below

```
const generateInvoiceNumber = async () => {
        // Retrieve the number of invoices in the database
        // This count can be used to generate an invoice number as invoices won't be deleted
        const collectionRef = collection(db, "Invoices");
        const snapshot = await getCountFromServer(collectionRef);

        // Get the next number
        let nextNumber = (snapshot.data().count).toString();

        //Make the number into the requrired 4 digit format
        while (nextNumber.length < 4) {
                nextNumber = "0" + nextNumber;
        }

        // Returns the formatted number
        return `INV${nextNumber}`;
};
```

#	Test	Pass / Fail	Evidence
11	With the current number of invoices in the database at 6, the next generated invoice number should be “INV0007”	FAIL	Invoice number # INV0006

Upon inspecting my code I noticed that I was retrieving the current number of invoices and using that as my number, rather than adding one to it. I edited the line below to be correct

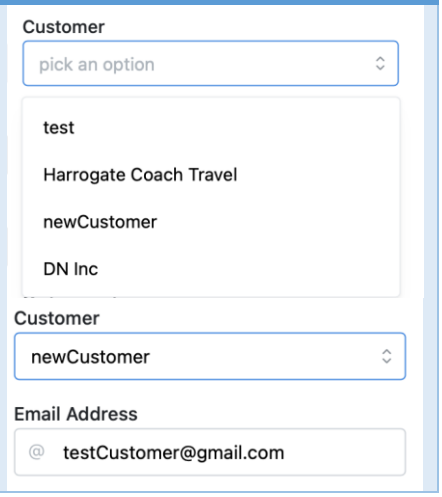
```
    let nextNumber = (snapshot.data().count + 1).toString();
```

#	Test	Pass / Fail	Evidence
11	With the current number of invoices in the database at 6, the next generated invoice number should be "INV0007"	PASS	

The ***populateCustomerDetails*** function automatically populates the email field when a customer has been selected on the create invoice page. It is called in ***CreateUpdateInvoice.jsx*** when a customer has been selected.

```
// Get the selected customer's email from the database
const populateCustomerDetails = async (selectedCustomer) => {
        const customerRef = doc(db, "Customers", selectedCustomer);
        const docSnap = await getDoc(customerRef);

        // Document has been retrieved
        if (docSnap.exists()) {
                // Populate the email field
                const doc = docSnap.data();
                setEmail(doc.email);
        } else {
                // Document wasn't found
                console.error("document not found");
        }
};
```

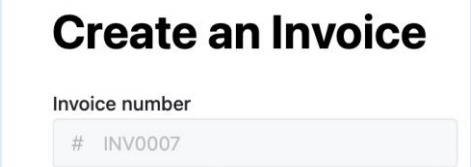
A scenario where a customer document isn't found cannot occur as the user can only select a value for ***selectedCustomer*** from a list that has been retrieved from the database when the page was loaded. This is only here in case of future expansion (I.e., adding a search field to the customer selection form)

#	Test	Pass / Fail	Evidence
12	When a customer is selected, the email field should be automatically populated with the customer's email address	PASS	

In ***CreateUpdateInvoice.jsx's useEffect*** hook, there are different functions that can be called based on whether the user is on the create or update page.

For create page specific actions, the ***createPageSetup*** function is called.

```
const createPageSetup = async () => {
        // Auto Populate the invoice number field
        const newInvoiceNumber = await generateInvoiceNumber();
        setInvoiceNumber(newInvoiceNumber);
};
```

Test	Pass / Fail	Evidence
When the create page is loaded, the Invoice number field will be automatically filled in.	PASS	

For update page specific actions, the ***updatePageSetup*** function is called.

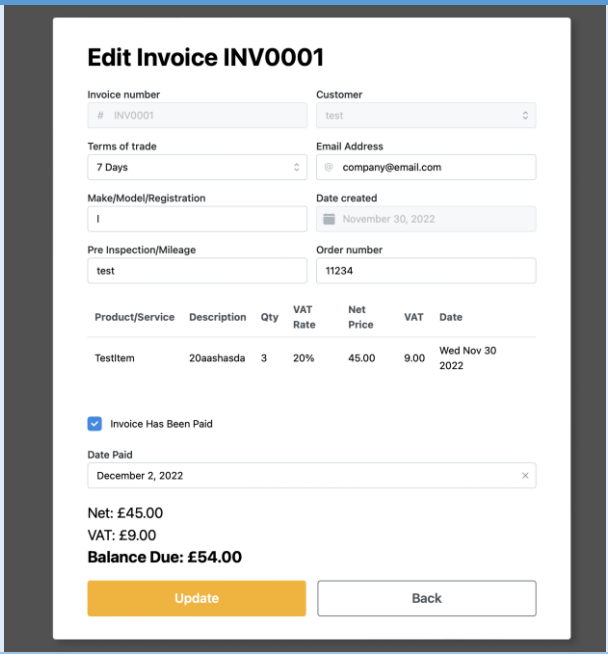
```
const updatePageSetup = async () => {
        const fetchedData = await getInvoiceInfo(invoice);
        if (fetchedData) {
                // The invoice exists
                const customerDoc = fetchedData.customer;

                // Check if the customer referenced still exists
                getDoc(customerDoc).then((docSnap) => {
                        if (docSnap.exists()) {
                                // The customer exists
                                setCustomer(customerDoc.id);
                        } else {
                                showNotification({
                                        title: "This invoice's customer is no longer in the database",
                                        color: "yellow.6",
                                        icon: <TiWarningOutline />,
                                });
                                setCustomer("Deleted");
                        }
                });
                // setCustomer(customerDoc.id);

                setInvoiceNumber(invoice);
                setTermsOfTrade(fetchedData.termsOfTrade);
                setEmail(fetchedData.email);
                setMakeModelReg(fetchedData.makeModelReg);
                setDateCreated(fetchedData.dateCreated.toDate());
                setPreInspection(fetchedData.preInspection);
                setOrderNumber(fetchedData.orderNumber);

                setInvoicePaid(fetchedData.invoicePaid);
                // Only set the date paid if the invoice has been paid
                if (fetchedData.invoicePaid)
                        setDatePaid(fetchedData.datePaid.toDate());
```

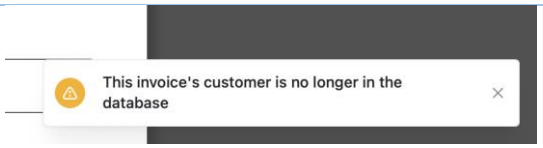
The code in blue sets the state for each field in the invoice form.

(code continues below)

```
                // firebase stores dates in a different format to how Mantine reads them
                // So each invoice item must be formatted with the toDate() method
                // Same goes with the dateCreated field above
                let formattedItems = [];
                fetchedData.invoiceItems.forEach((item) => {
                        const newItem = {
                                ...item,
                                date: item.date.toDate(),
                        };
                        formattedItems.push(newItem);
                });
                setInvoiceItems(formattedItems);

                // Get the net price for the invoice by parsing the invoice items fetched from the database
                const netPrices = fetchedData.invoiceItems.map(
                        (item) => item.price
                );
                const sumNetPrices = netPrices.reduce((acc, val) => acc + val);
                setNetPrice(sumNetPrices);

                // Get the vat price from the invoice items fetched from the database
                const vatPrices = fetchedData.invoiceItems.map((item) => item.vat);
                const sumVatPrices = vatPrices.reduce((acc, val) => acc + val);
                setVatPrice(sumVatPrices);
        } else {
                // The invoice does not exist
                notFoundNotification("invoice");
        }
};
```

notFoundNotification is a custom function that displays “xxxx could not be found”

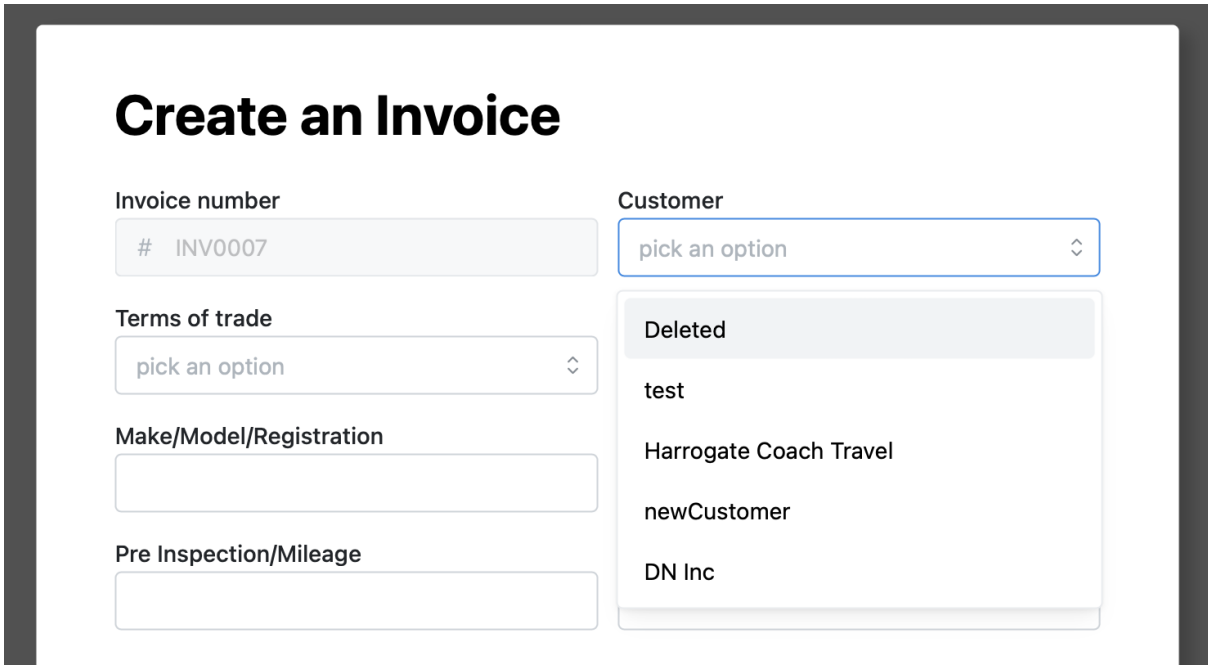
Test	Pass / Fail	Evidence
When the invoice and its associated customer both exist, all the invoice’s fields are automatically fetched from the database and filled in.	PASS	

When the invoice exists but its associated customer has been deleted, a warning notification is displayed.	PASS	
When the invoice doesn't exist, an error notification is displayed.	PASS	

For general actions that run on both the create and update pages, the ***generalActions*** function is called.

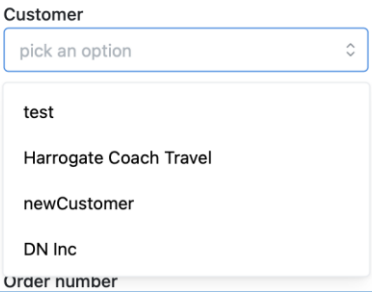
```
// Setup actions when the create page is loaded
const generalActions = async () => {
        // Get the data for the customers selection field from the database
        const customersQuery = query(collection(db, "Customers"));
        const querySnapshot = await getDocs(customersQuery);

        // Field that displays when an invoice's customer has been deleted
        let customers = ["Deleted"];
        querySnapshot.forEach((doc) => {
                customers.push({ label: doc.data().companyName, value: doc.id });
        });
        setCustomerData(customers);
};
```

The way I am alerting the user that the customer has been deleted is by adding a value field to the customerData state (this controls the inputs for the customer selection field). This is fine to do as the customer selection field is uneditable on the update page. But upon further testing, I discovered that this deleted field was accessable on the create page as well (shown below).



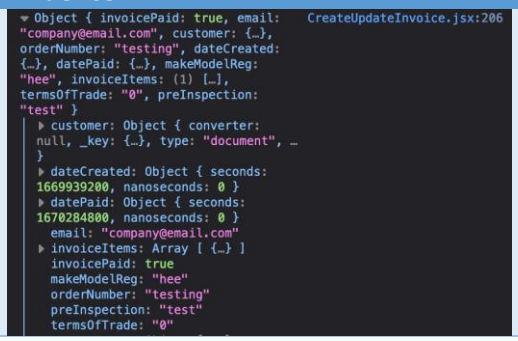
To fix this, I edited the line of code responsible for creating this value to only create it on the update page.

```
// Field that displays when an invoice's customer has been deleted if the page is set to update but not create
let customers = action === "update" ? ["Deleted"] : [];
```

Test	Pass / Fail	Evidence
The customers should be accessible on the create page and the "Deleted" value should only be created on the update page (not visible on the create page when a customer is being selected)	PASS	

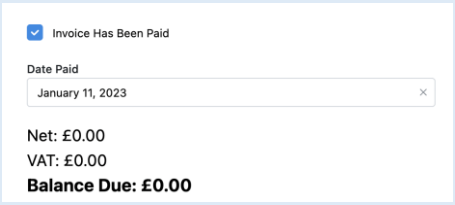
The ***getInvoiceInfo*** function is responsible for fetching the required invoice from the database.

```
// Retrieve the inputted invoice's data from the database (effectively a getter function)
const getInvoiceInfo = async (invoiceNumber) => {
        const docRef = doc(db, "Invoices", invoiceNumber);

        const docSnap = await getDoc(docRef);

        if (docSnap.exists()) {
                // The invoice has been found
                return docSnap.data();
        } else {
                // The invoice doens't exist
                return false;
        }
};
```

	Test	Pass / Fail	Evidence
6a	The function should return the invoice when a valid invoice number is inputted	PASS	
6b	The function should return false when an invalid invoice number is inputted	PASS	

I also added the ***invoicePaid*** and ***datePaid*** fields to the invoice form.

```
<SimpleGrid cols={1} span={8}>
        <Checkbox
                label="Invoice Has Been Paid"
                checked={invoicePaid}
                onChange={(e) => {
                        setInvoicePaid(e.target.checked);
                }}
        />
        {/* Dropdown field only shown when invoicePaid checkbox has been ticked */}
        <Collapse in={invoicePaid}>
                <DatePicker
                        label="Date Paid"
                        value={datePaid}
                        onChange={setDatePaid}
                        // Cannot have paid the invoice before it was created
                        minDate={dateCreated}
                        // Cannot have been paid if date is after today
                        maxDate={new Date()}
                />
        </Collapse>
</SimpleGrid>
```

Test	Pass / Fail	Evidence
The datePaid field should only be visible when the invoicePaid checkbox is checked	PASS	Checked  Unchecked 

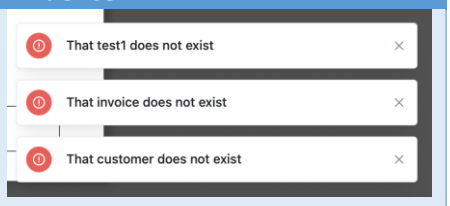
For the ***datePaid*** field, I have set a minimum date of when the invoice was created, as the invioice cannot have been paid before it was created. There is also a maximum date of the current date the user is on the website, as the invoice cannot have been paid if the date paid hasn't happened yet. I have applied this maximum date to the dateCreated field as well (shown below)

```
<DatePicker
        label="Date created"
        icon={<BsFillCalendarFill />}
        value={dateCreated}
        onChange={setDateCreated}
        disabled={action === "update"}
        // Invoice cannot have been created in the future
        maxDate={new Date()}
/>
```

I have created a general function to show a notification that "xxxxxx couldn't be found" called ***notFoundNotification***

```
export const notFoundNotification = (fieldName) => {
        showNotification({
                title: `That ${fieldName} does not exist`,
                color: "red",
                icon: <BiErrorCircle />,
        });
};
```

Test	Pass / Fail	Evidence

Input various text to make sure the notification displays correctly
 	
`notFoundNotification("test1");`
`notFoundNotification("invoice");`
`notFoundNotification("customer");`
	PASS	

I also added functionality to the ***<HomeButton />*** component

```
const HomeButton = ({ size }) => {
        const navigate = useNavigate();
        const handleClick = () => {
                navigate("/");
        };
        return (
                <Button
                        fullWidth
                        size={size || "lg"}
                        color="gray.7"
                        variant="outline"
                        onClick={handleClick}
                >
                        Home
                </Button>
        );
};

export default HomeButton;
```

******Tests that haven't been numbered are functions that weren't accounted for in design******

Evaluation

Post-Development (Evaluative) Testing

System Functionality Tests

Test Number		Explanation	Test Data	Expected Result	Pass/Fail
1		When creating a customer, there is a maximum character limit for each field	No specific data inputted, just spam letters into each field to test how many characters each field allows.	After a certain length, each field will stop editing the input as the maximum length has been reached	PASS
2	A	When submitting a customer creation form, the client puts the inputted email address and mobile number through validation function to check that they are in the correct format.	**For email field:** testCustomer@gmail.com	-	PASS
	B		customerEmail1@test.co.uk	-	PASS
	C		"notAnEmail"	-	PASS
	D		**For mobile field:** "01924 251 222"	-	PASS
	E		"0800 122 223"	-	PASS
	F		" " (empty form)	-	PASS
3		Test the entire workflow of creating a customer. With the software being used as intended	No specific data inputted, just start on the home page and work through creating a customer on the system from start to finish.	The customer will be created and the user will receive a notification alerting the customer has been created.	PASS
4		Test the entire workflow for creating an invoice. With the software being used as intended	No specific data inputted, just start on the home page and work through creating a customer on the system from start to finish.	The invoice will be created and the user will receive a notification alerting them of this	PASS
5		Test the process of logging in and out if the software using a google account.	No specific data inputted, just start on the logged out page and sign in with a Gmail account, then sign out again	The user will see the home page when they log in and get a	PASS

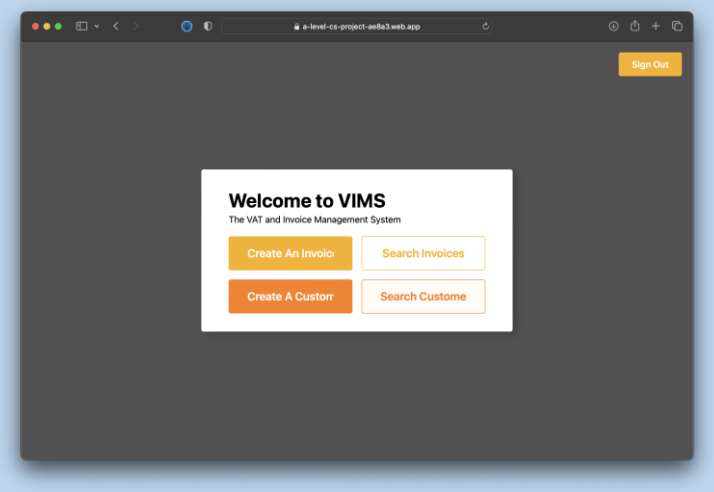
				notification when they sign out	
6	A	Test the entire workflow of updating / deleting a customer	The user will start on the homepage and work through updating a specific field on a customer.	The user will get a notification to alert them for each stage	PASS
	B		The user will delete a customer.		PASS
7		Test the entire workflow of updating an invoice	No specific data inputted. The user will start on the homepage and work through updating an invoice's fields and paid status.	The user will receive a notification that they have successfully updated an invoice	PASS
8		Test that invoices behave correctly when their respective customer has been deleted	No specific data inputted. The user will view an invoice whose customer is no longer in the database.	The program will display "Deleted" in the customer input box and the user will get a warning notification.	PASS

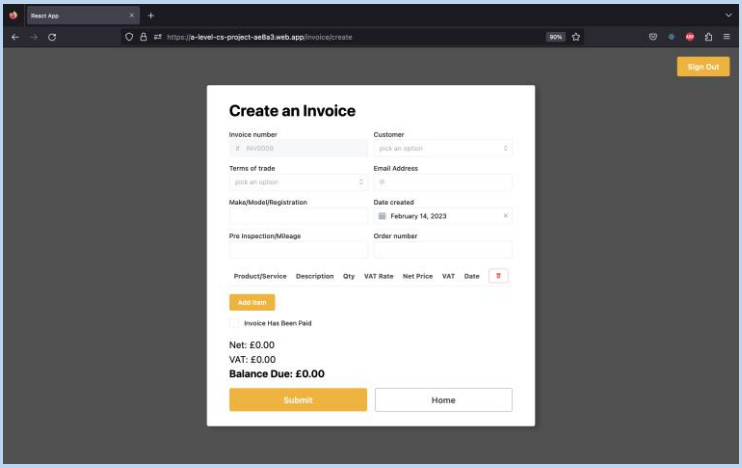
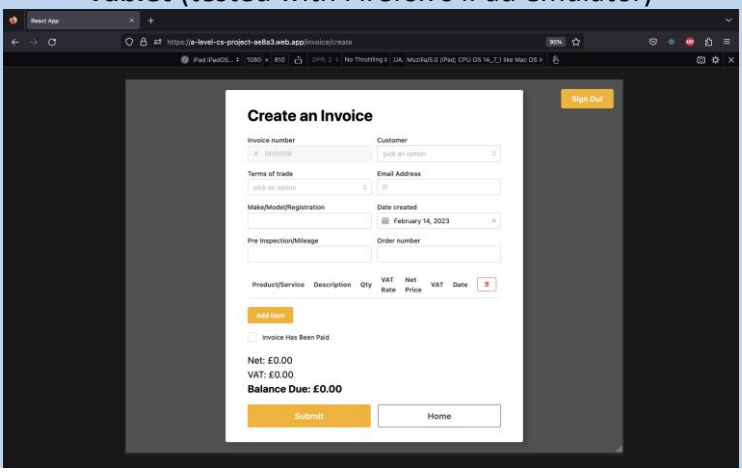
Test Number	Evidence
3	Project Demo 0:22 – 1:07
4	Project Demo 1:07 – 2:30
5	Project Demo 0:00 – 0:22
6a	Project Demo 2:30 – 2:55
6b	Project Demo 2:55 – 3:04
7	Project Demo 3:04 – 3:20
8	Project Demo 3:20 – 3:31

All tests refer to video material uploaded to YouTube (https://youtu.be/Kv01_omOtLY) Links in the table jump directly to required time

Evidence for 1 and 2 cannot be given as they do not display a visible output

Success Criteria Tests

Test Number	Description	Pass/Fail	Evidence
1	Invoices will be added to the database within 5 seconds of clicking submit	PASS	
2	Invoice items must show a **real-time** net/vat/total	PASS	Shown in Project Demo (1:47 – 1:56)
3	Invoices and Customers should load to search table within 3 seconds of page loading	PASS	Invoice table. Customer Table 
4	Customers should be deleted from the database within 3 seconds of the delete button being pressed	PASS	
5	The UI should display on desktop PC, laptop, and landscape iPad screen.	PASS	**Laptop** 

			PC  **Tablet** (tested with Firefox's iPad emulator) 

Success of the Solution

I will measure my solution's success by evaluating the success criteria (see analysis and the testing table above), and receiving stakeholder feedback.

Stakeholder feedback was received through a custom designed questionnaire as shown below.

VIMS stakeholder survey

(give ratings from 1 to 10 and a brief explanation)

1) **The website was intuitive to navigate when you first picked it up**

   (1 – Strongly Disagree 10 – Strongly agree)

---

2) **The website aesthetically pleasing and consistently coloured**

   (1 – Strongly Disagree 10 – Strongly agree)

---

3) **The website was easy to use without a guide**

   (1 – Strongly Disagree 10 – Strongly agree)

---

4) **How fast did the website load/submit data?**

   (1 – Strongly Disagree 10 – Strongly agree)

---

5) **How happy are you with the solution as a whole**

   (1 – Very unhappy 10 – Very Happy)

The survey was filled out as a collaborative effort by both my primary and secondary stakeholders as different parts of the solution are better suited to each of them. This gave me a more comprehensive view on the solution's success as a whole.

VIMS stakeholder survey

(give ratings from 1 to 10 and a brief explanation)

1) **The website was intuitive to navigate when you first picked it up**

   (1 – Strongly Disagree 10 – Strongly agree)

   10 – There is no need for instructions as the big buttons on the VIMS home page are clearly labelled re their purpose, making the website extremely easy to navigate around.

---

2) **The website aesthetically pleasing and consistently coloured**

   (1 – Strongly Disagree 10 – Strongly agree)

   10 – There is no jargon or unnecessary detail on the website which makes it extremely easy to navigate. The colours used reflect some of the colours in our business logo and this is pleasing to see.

---

3) **The website was easy to use without a guide**

   (1 – Strongly Disagree 10 – Strongly agree)

   10 – As said above, the home page buttons clearly indicate their purpose. When clicking through the buttons to the pages that sit behind the home page, these are also self-explanatory re how to fill them in. The content of each web page is exactly as we had requested and contains all the detail we need in a format that is easy for the user to complete.

---

4) **How fast did the website load/submit data?**

   (1 – Strongly Disagree 10 – Strongly agree)

   10 – The website loaded almost as soon as it had been launched and similarly, when testing, the information that we input into the new customer and new invoice pages saved straight away. The search options also loaded the stored data without delay.

---

5) **How happy are you with the solution as a whole**

   (1 – Very unhappy 10 – Very Happy)
   10 – The brief has been fully followed and provides us with exactly what we need. This software will prove extremely useful for our business going forward. Our thanks to Daniel for creating this solution for us.

Usability Features

Website Navigation

Website navigation was thoroughly tested throughout development purely by using the website. For evidence of navigation of the site refer to the Project Demo as a whole (site navigation is used throughout each workflow demonstrated).

Stakeholder feedback (see ***success of the solution***) suggests that the the website is intuitive to navigate as expected. With buttons on the home page described as *"clearly labelled".*

Consistent theming

As described in the design stage, Buttons are colour-coded based on their function. I followed my planned theming exactly as described in design, which can be seen across the Project Demo.

Again, referring to the stakeholder feedback survey, the website has been described as *"self-explanatory"* with the colours chosen reflecting some of those from their business logo, giving the solution a personal touch and cementing its bespoke nature.

Notifications System

As shown in design, I had plans to implement a notifications system that came with MantineUI and this was implemented perfectly. As shown in the Project Demo when creating/updating/deleting data from the database - notifications are displayed to the user whenever a server request is made, to show the user that they have clicked the button.

Limitations and Maintenance

As expected, my solution isn't perfect and has some limitations. Some of these can be addressed with further development of the solution and others stem from external reasons outside of my control.

Firstly, the software cannot format data into emails and send them to customers. This means that my stakeholders will have to manually copy data from the website and fill it into an email. This could become quite a tedious process if they have a large batch of invoices to complete for different customers. As mentioned in analysis, this could be implemented with further development but - based on the time allocated for this project – was unfeasible as email APIs cost money to use and formatting software must be learnt how to use with limited adaptability if my stakeholders change how they want the invoice PDF to look.

Another limitation that was unforeseen is that invoices cannot be deleted from the database. This is due to the way that the invoice numbers are generated. The system looks at how many invoices already exist in the database and add one to that number to get the next invoice number. If invoices were deleted, there would be duplicate invoices which could be a major limitation when customers ask for copies of invoices from long periods of time ago, and my stakeholders cannot tell which is which. This limitation isn't an issue now but in a couple of years when there are thousands of

invoices clogging up the online storage, some will want to be archived to save money on hosting the webpage.

In the future, maintenance will need to be carried out on the system for it to continue functioning. For example, google could update their firebase SDK's syntax to a new version (currently version 9). This will not break the system as for a long period of time after version 10 is released, version 9 will continue to be supported. The only reason the code would need to be updated is to gain access to new features that could be released making development easier.

Also, the standard VAT rate will need to be updated if the government raise or lower it in the future. This is currently a hard-coded value that must be changed by hand. In the future, I could develop the program to parse the government's website for the current VAT rate and use that rather than a hard-coded value, so it updates automatically.

Because the program is written in JavaScript, it relies heavily on third party dependencies which could stop being supported at any time. This would require me to find a new alternative and potentially refactor all functions in the codebase that related to the previous dependency to match with the syntax of the new one. This is made easier through how modular I made the code, each function can essentially be isolated and run by itself at a base level which makes transition between dependencies easier as the code can be converted in chunks whilst still functioning rather than needing to be done all at once.

--- END OF REPORT ---