University of Southampton

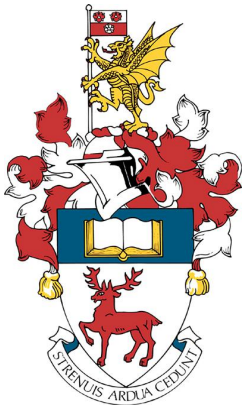
Electronics and Computer Science

COMP6251: Web & Cloud App Development

React Web Application: Tenner

Charles Powell (cp6g18, 29970245), Ben Lumsden (bjl1g19, 30901677)
Alex Burton (azsb1g19, 30740673), Thomas Hoad (tdh1g19, 30914515)

May 18, 2023



Mark Distribution

Group:	1

Student ID	Email ID	Percentage	Signature	Date
30740673	azsb1g19	25	Alex Burton	18/05/2023
30901677	bjl1g19	25		18/05/2023
29970245	Cp6g18	25	Charles Powell	18/05/23
30914515	Td1g19	25	Thomas Hoad	18/05/23

Contents

Preface i

Mark Distribution i
Contents ii

Report 1

1 Introduction 1

2 Technical Details 3
2.1 Tech Stack . . . 3
2.1.1 Front-End . . . 3
2.1.2 Back-End . . . 3
2.1.3 Programming Language . . . 3
2.2 APIs . . . 3
2.3 Authentication . . . 5

3 Description of System 5
3.1 Basic Requirements . . . 5
3.1.1 Authentication . . . 5
3.1.2 Admin . . . 6
3.1.3 Provider . . . 6
3.1.3.1 Provider Profile . . . 6
3.1.3.2 Search Services . . . 7
3.1.3.3 Create and Edit Services . . . 7
3.1.3.4 Provider Order History . . . 8
3.1.3.5 Provider Notifications . . . 8
3.1.4 Customer . . . 8
3.1.4.1 Customer Profile . . . 8
3.1.4.2 Customer Order History . . . 9
3.1.4.3 Searching Services . . . 9
3.1.4.4 Viewing Service Reviews . . . 9
3.1.4.5 Ordering Services . . . 9
3.1.4.6 Providing Additional Details . . . 9
3.1.4.7 Providing Reviews . . . 10
3.1.4.8 Customer Notifications . . . 10
3.2 Advanced Requirements . . . 10
3.2.1 Social Media Logon . . . 10
3.2.2 Cloud Hosting . . . 11

4 Testing 11
4.1 Business Logic . . . 11
4.1.1 Profile Page . . . 11
4.1.2 Orders Page . . . 12
4.1.3 Service Listing Page . . . 12
4.1.4 Header . . . 12
4.2 Compatibility . . . 12
4.3 Input Checking . . . 12
4.3.1 Authentication . . . 13
4.3.2 New Service . . . 13
4.3.3 New Service Request . . . 14
4.3.4 Orders Modals . . . 14
4.4 Firebase Rules . . . 14

SECTION 1

Introduction

This report presents the implementation of a web application called 'Tenner'. The application allows for individuals to register as either a customer or provider, and purchase or request common freelance services such as babysitting, cleaning, pest control and much more.

The basic structure of the application can be divided into the following sections, which are referenced throughout the remainder of this report.

- **Launchpad**: A Home Screen for the application that showcases the categories of services on offer to customers, as well as some trending services.

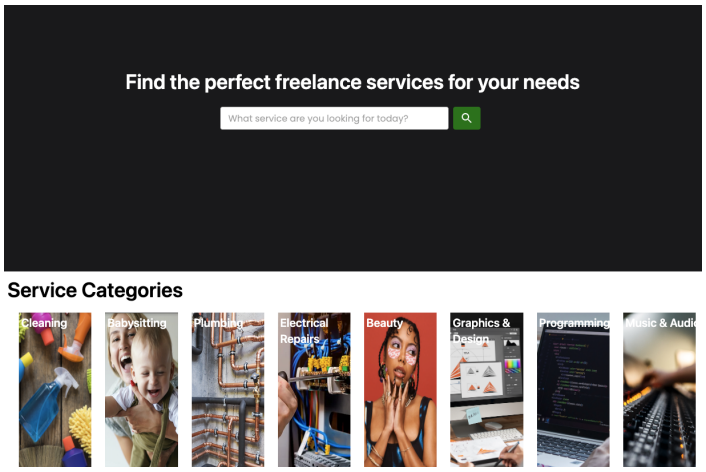


Fig. 1.1. Launchpad

- **Header**: An element bar that remains at the top of the application across all pages and provides navigation to its main pages. The header also updates dynamically based on the authentication status of the user, and provides a search bar to allow the user to search for services from anywhere in the application.

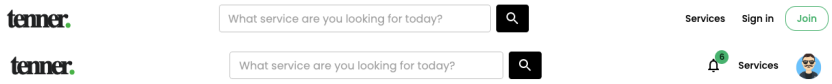


Fig. 1.2. Header for logged out and logged in user

- **Services Page**: A page for browsing through the complete set of services available on the platform.

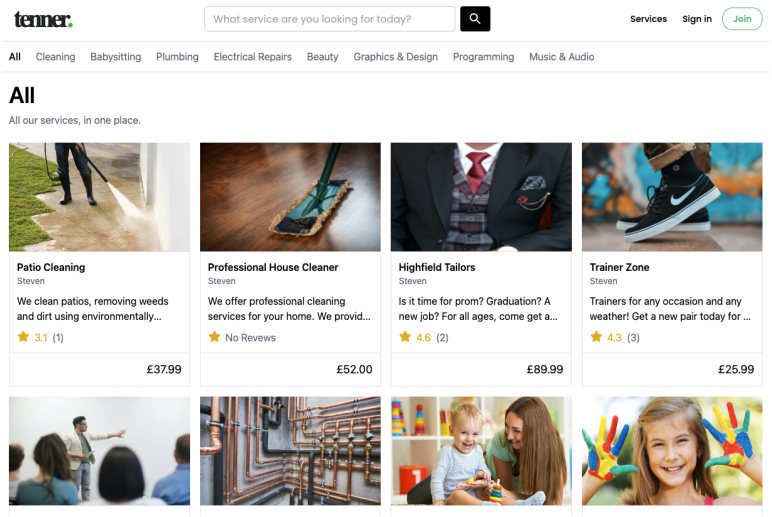


Fig. 1.3. Services page

- **Service Listing Page**: A page for inspecting an individual service in detail, as well as purchasing this service.

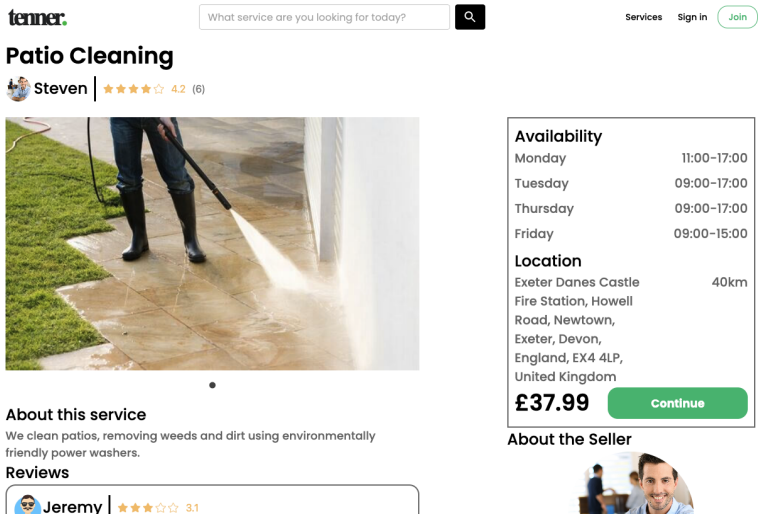


Fig. 1.4. Service Listing page

- **Profile Page**: A page for viewing account information and order history. Customers and providers are shown different profile pages which tailor the displayed information to the workflow of the user type.

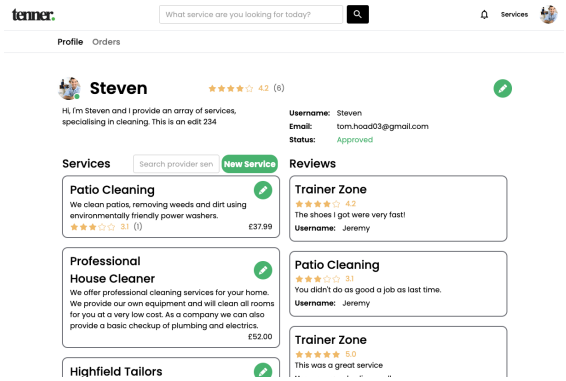


Fig. 1.5. Provider profile page

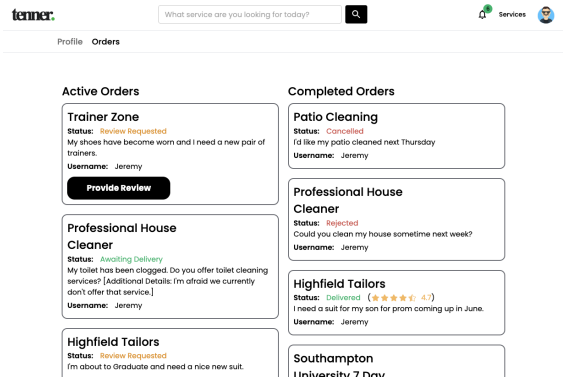


Fig. 1.6. Customer profile page

- **Input Modals**: A number of modals are used throughout the application to handle user input, for example, in the case of registration/sign-in and order submission.

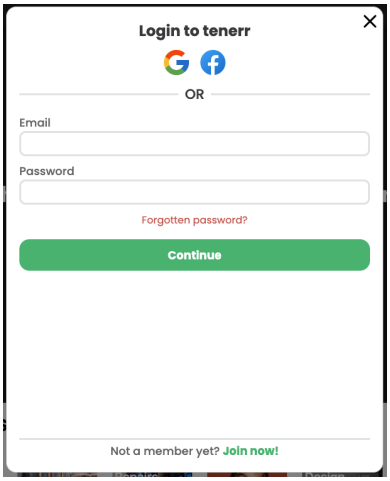


Fig. 1.7. Authentication input modal

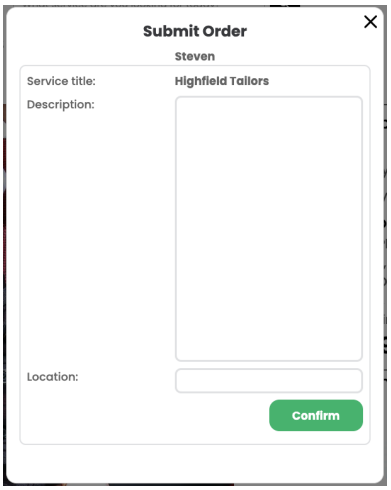


Fig. 1.8. Order input modal

SECTION 2

Technical Details

2.1 Tech Stack

The following descriptions summarise the key technologies used throughout the various components of the Tenner web application.

2.1.1 Front-End

The front end of the application is implemented using React. React was chosen as it provides a component based architecture, state management, efficient rendering, and many more features.

A number of common front-end libraries were also used to handle specific parts of the front-end experience. The most prominent of these are:

- `react-router`: Handles the navigation between the application's various pages.
- `material-ui`: Provides pre-built React components to support efficient development process and attractive front-end.
- `tailwindcss`: Provides pre-configured css styling in a convenient format to support an efficient development process and attractive front-end.

2.1.2 Back-End

Firebase is used for the back-end of the application. Specifically, Firebase's Realtime Database and Authentication services are used to handle data storage and authentication for the application respectively. Firebase was chosen as it is a no-code solution, automatically handles numerous technical aspects such as security, has excellent documentation and integrates well with React.

2.1.3 Programming Language

Typescript is used as the primary programming language throughout the application as it provides type safety and enhanced error detection which allows for a more efficient development process. Typescript also allows for the definition of custom datatypes for the data returned by the application back-end, providing a more robust application workflow and ensuring that the front-end application is concerned only with the display of data, and not its processing.

2.2 APIs

The system consisted of various application programming interfaces (APIs) to integrate the back end, from Firebase, with the front end. As Firebase Realtime database returns data as a JSON, the use of APIs allowed types to be defined for the expected return time. This enforced type safety with the front end when handling data from the back end; mitigating the potential of errors.

Listing 1: Example data types of data handled from Firebase Realtime Database

```
interface Time {
  hour: number
  minute: number
}
export interface TimeRange {
  startTime: Time
  endTime: Time
}
export enum Day {
  'Monday',
  'Tuesday',
  'Wednesday',
  'Thursday',
  'Friday',
  'Saturday',
  'Sunday'
}
export type Availability = Partial<Record<Day, TimeRange[]>>
export interface LocationData {
  name?: string
  lat?: number
  lng?: number
  r?: number
}
```

```
export interface Service {
  id: string
  name: string
  providerID: string
  description: string
  pictures?: string[]
  price: number
  category: ServiceCategory[]
  availability: Availability
  location?: LocationData
}
```

To ensure the code remains modular and robust to changes in data types, four APIs were created:

- auth.ts - Defines type for an account status, and provides functions to register users, sign in, sign in with Google/-Facebook, reset password and sign out.
- notifications.ts - Defines the Notification data type and provides functions to create a new notification, remove a notification, get all notifications for a user, and a hook to listen to notification changes for a user.
- services.ts - Defines the data type for a Service, and all associated types. The API also provides the following functions:
  - Creating, removing, updating and loading services.
  - Creating and updating service requests.
  - Creating, updating and loading service reviews.
  - Finding the trending services.
  - A hook to listen to changes in services.
- user.ts - This API provides a type definition for the user type, as well as functions to create a new user, upload profile pictures, update a user's details, load a given, or currently authenticated, user's details, and a hook to listen to changes in a given user's details.

As the data returned from the Firebase Realtime database is asynchronous, promise-based asynchronous programming techniques were employed, so that the returned data can be handled as if it is synchronous on the front end by chaining asynchronous methods and handling errors in a central body stream.

Listing 2: Promise-based asynchronous programming to add a service to Firebase Realtime Database

```
export const addService = async (service: NewServiceParam) => {
  const currentUser = auth.currentUser
  const db = getDatabase()
  const newServiceRef = push(child(ref(db), 'Services')).key

  return await new Promise<Service>((resolve, reject) => {
    if (currentUser === null) {
      reject(new Error('No current user authenticated'))
    } else if (newServiceRef === null) {
      reject(new Error('Failed to create service request key'))
    } else {
      set(ref(db, `Services/${newServiceRef}`), {
        providerID: currentUser.uid,
        ...service
      })
        .then(() => {
          console.log(`Successfully created new service with key: ${newServiceRef}`)
          resolve({
            id: newServiceRef,
            providerID: currentUser.uid,
            ...service
          })
        })
        .catch(err => {
          reject(err)
        })
    }
  })
}
```

However, as Promises only handle one-time aynshronous operations, they are unable to handle data which changes in real time. Instead, Firebase listeners were employed to listen to changes in the Realtime database data. These listeners invoke a callback function when the data changes. As the API is contained in a separate file to where the method to listen to changes in data may be invoked, state management and hooks are used to handle this data. According to React documentation, the name for all methods returning a hook must start with the 'use' keyword. The listing below shows an example of the *useGetNotifications* hook, which listens to changes in the notifications for a user. In this code, the method returns a hook containing the user's notifications. A *useEffect* is used, so that if the user changes then the

reference in the database, the point at which to listen to changes in data, is changed. The *unsubscribe* function listens to changes in the data at the given database reference, and updates the user's notifications hook accordingly. By returning the *unsubscribe* method in the *useEffect*, the function is cleaned up once the component calling the hook unmounts. I.e, the API will stop listening to the data at that database reference. This prevents any unnecessary data fetching.

Listing 3: Handling data streams in the API

```
export const useGetNotifications = (user: User) => {
  const [notifications, setNotifications] = useState<NotificationProps[]>([])
  const db = getDatabase()

  useEffect(() => {
    const loadData = () => {
      const notificationsRef = ref(db, `Users/${user.type}/${user.id}/Notifications`)
      const unsubscribe = onValue(notificationsRef, (snapshot: any) => {
        const notificationsData = snapshot.val()
        const notifications: Promise<NotificationProps>[] = []

        for (const id in notificationsData) {
          const notification = notificationsData[id]

          if (!notification.serviceRequestID) {
            notifications.push({ notificationID: id, user: user, ...notification })
          } else {
            const serviceRequestPromise = getServiceRequest(notification.serviceRequestID)
              .then(result => {
                return { notificationID: id, user: user, serviceRequest: result, ...↩
                    ↪notification }
              })
              .catch(err => Promise.reject(err))

            notifications.push(serviceRequestPromise)
          }
        }

        Promise.all(notifications).then(result => setNotifications(result))
      })
      return () => {
        unsubscribe()
      }
    }

    loadData()
  }, [user])

  return notifications
}
```

Overall, the usage of APIs has encouraged consistency between the front end and back end, whilst promoting suitable handling of asynchronous and real time data streams.

2.3 Authentication

Authentication for the system is implemented using Firebase Authentication. The service hosts the ability to authenticate using multiple authentication providers. For this project, email/password, Google, and Facebook were used to administer authentication to a user. The main advantage of using Firebase authentication is that it securely stores the user's email address and user IDs. The user IDs can be used to store additional details for the user in the Realtime Database, whilst protecting their email address from unauthorised access. Furthermore, upon registering a user, Firebase Authentication provides built-in error handling to ensure that the account does not already exist and that the password is strong. Firebase Authentication can also be integrated with the Firebase Realtime Database rule set, to ensure that an authenticated user can only access the correct data which they are authorized for. This ensures the system remains secure and robust.

Section 3
Description of System

3.1 Basic Requirements

3.1.1 Authentication

As previously discussed, the user is authenticated using Firebase Authentication. On the front end, the authentication status is handled in the application header, using the *onAuthStateChanged* handler provided by Firebase Authentication.

If a user is authenticated, then the header will change to show account menu buttons. Otherwise, the headers shows buttons to login or register.

Listing 4: Managing the authentication status of a user

```
useEffect(() => {
    const auth = getAuth()
    const unsubscribe = onAuthStateChanged(auth, user => {
      if (user) {
        getUser(user.uid)
          .then(result => setUser(result))
          .catch(err => {
            console.log(err)
            setUser(null)
            setAuthModalState('WELCOME')
          })
      } else {
        setUser(null)
      }
    })
    return unsubscribe
}, [authModalState])
```

To create a new account, the user is able to register using email/password or social media logon. Upon successfully creating a new account, the user is welcomed in a new modal which allows the user to set up their personal details. This involves choosing a username, a profile picture, and their role. In this case, their role can either be Customer or Provider. If the user chooses to be a Provider then they must enter a description of themselves.

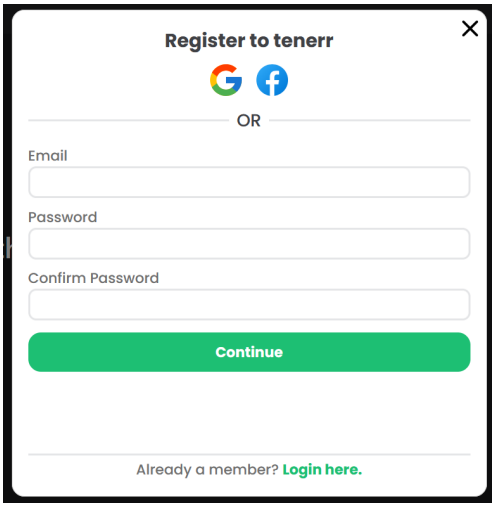


Fig. 3.1. Register modal for a new user

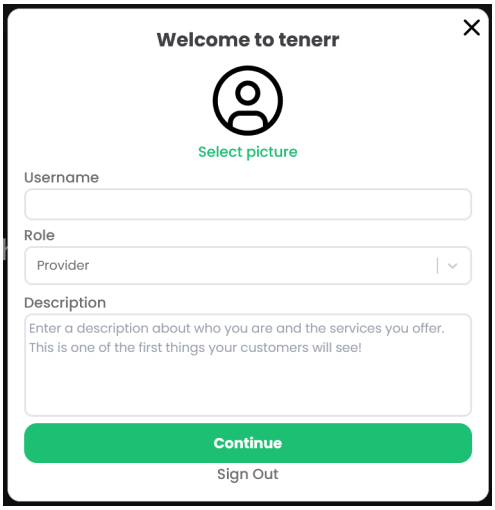


Fig. 3.2. Welcome modal for a new user

A similar modal allows the user to login with their details, with appropriate error checking provided by the Firebase Authentication SDK to ensure the password is correct and the email exists.

3.1.2 Admin

Upon registration, the user's account status will be 'Pending'. By interfacing with the Firebase console, the admin is able to update the user's status to 'Approved', 'Rejected' or 'Request'. This allows the admin to request more details from the user by adding a 'requestDetails' field, with the information that the admin would like to receive. Furthermore, the admin is able to delete reviews through the Firebase console.

3.1.3 Provider

The service provider's role is to create services for customers to request. They can create multiple services and can receive reviews from customers about service requests. A provider can control and view most of these capabilities from their profile. Customers should also be able to view much of this information on an admin-approved provider's profile.

3.1.3.1 Provider Profile

The filled-out provider profile looks a little different to the customer profile with services and reviews now shown. A provider profile is there to advertise specific services for a provider and showcase a breakdown of reviews to customers. All users can use a search bar to filter the services setup. The provider themselves can use the profile page to create new services and edit existing ones. The provider profile also displays an average star rating for the overall profile, each individual service and for every review. The provider profile page is showcased in Fig. 3.9.

3.1.3.2 Search Services

A key part of the services section is a search bar for all users to filter the provider services. Whilst the launchpad and services page both provide a significant amount more filtering and searching capability, this search function will automatically filter services by the query without the page reloading.

Listing 5: Searching services by hiding components that don't match the query.

```
const searchChange = () => {
  const serviceDivs = document.querySelectorAll("[class^='serviceDiv']")
  const searchBar = document.querySelector<HTMLInputElement>('#text-field')

  for (let i = 0; i < serviceDivs.length; i++) {
    const element = serviceDivs.item(i)
    if (element !== null) {
      const serviceDiv = document.getElementById(element.id)
      if (serviceDiv !== null && searchBar !== null) {
        const searchValues = searchBar.value.split('␣')
        for (const searchValue of searchValues) {
          if (serviceDiv.id.toString().toUpperCase().includes(searchValue.toUpperCase())
              ↪&& searchValue !== '') {
            serviceDiv.style.display = 'flex'
            break
          } else if (searchValue != '') {
            serviceDiv.style.display = 'none'
          } else {
            serviceDiv.style.display = 'flex'
          }
        }
      }
    }
  }
}
```

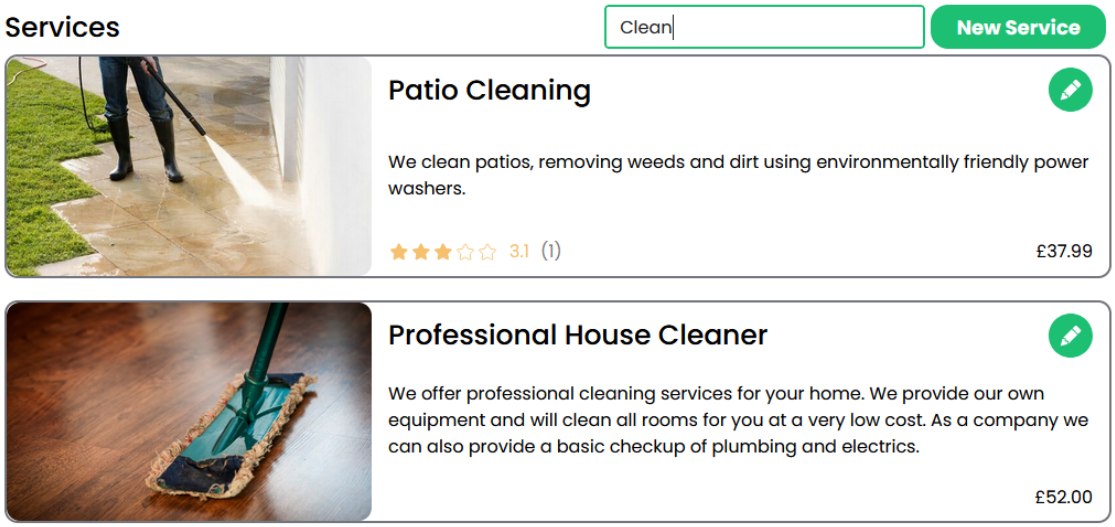


Fig. 3.3. Filtered Services

3.1.3.3 Create and Edit Services

Approved providers can create new services through a modal using the 'new service' button. This modal lets a service have a title, description, category, availability, location, images and price. The availability and location have their own modals that integrate into the main service modal. We provide an option to create new services and edit existing services which both use the same modal layout but with different functionality behind the scenes. The availability modal lets providers select the days and times of the week the service is available. They can choose any number of days and any time of day to be available. The location modal lets the provider set a service location and working radius by using the provided map.

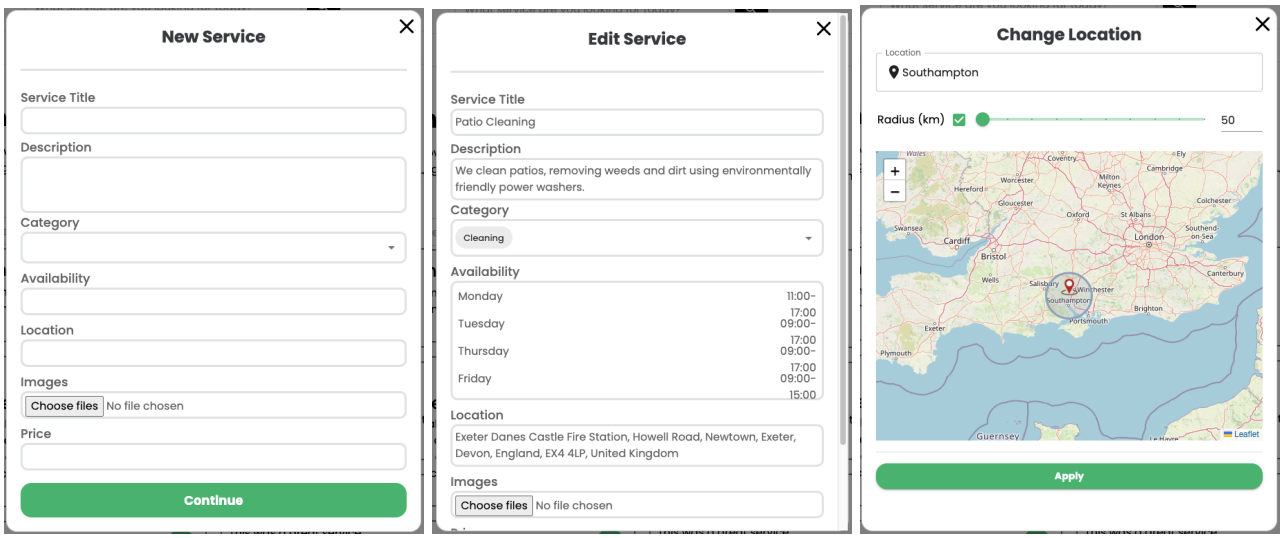


Fig. 3.4. New/Edit Service Service Modals

3.1.3.4 Provider Order History

Similarly to the customer, the provider can also see the order history page. On this page, providers can also see their active and completed orders but are provided with a different set of controls and information. Providers have the ability to approve, reject, deliver, cancel and request more details from customers. In the completed orders, the review will update with a star rating when the customer has provided one. When a customer has provided more details, the provider will have an option to further request details or cancel the order in addition to being able to deliver. These give providers control over what service requests they want to accept and complete.

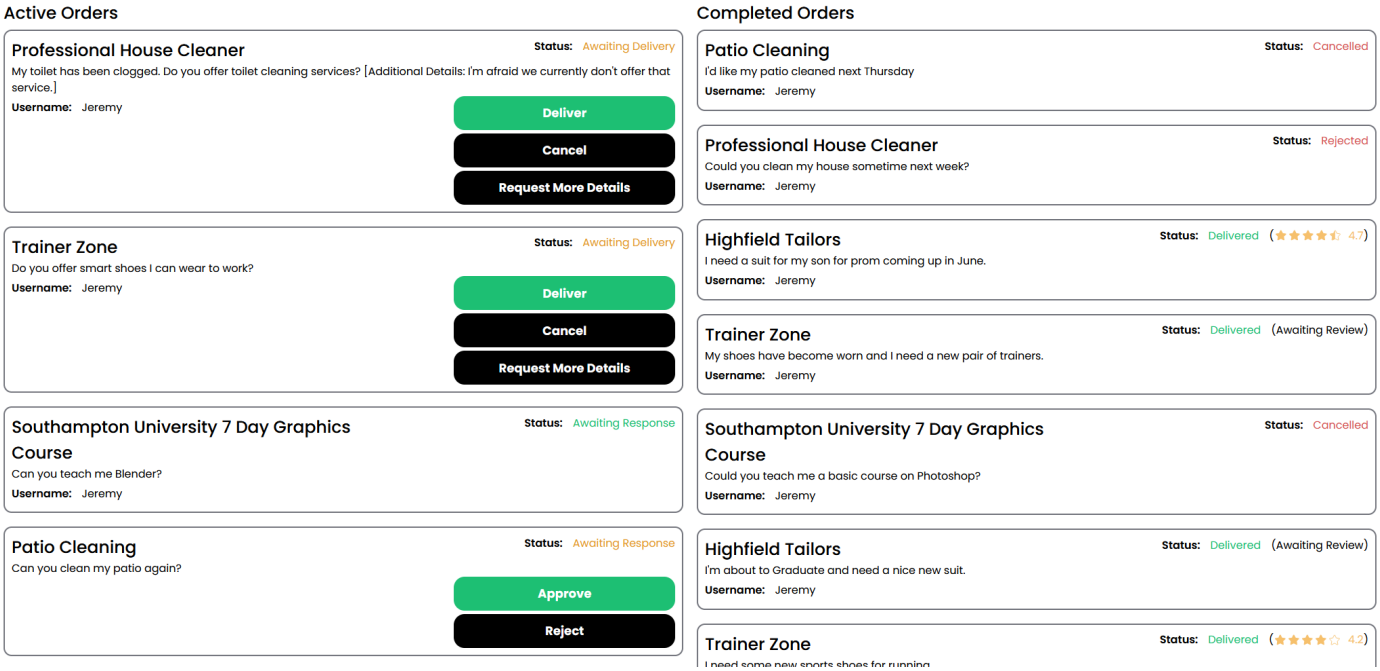


Fig. 3.5. Provider Order History page

3.1.3.5 Provider Notifications

Providers get notifications (similar to Fig. 3.11) for when a new order is placed, more information is given for an order, or when an order gets cancelled or reviewed.

3.1.4 Customer

3.1.4.1 Customer Profile

The customer profile and order history allows users to easily swap between the profile and order history pages using two buttons at the top of the page.

The customer profile contains personal information and settings for each user. This includes the username, profile picture, biography, and email address. An edit button is available for users to modify their username and profile picture. The profile page is depicted in the following figure:

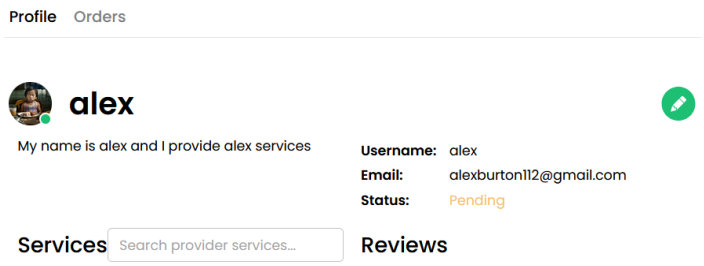


Fig. 3.6. Customer Profile page

3.1.4.2 Customer Order History

The order history (Fig. 3.10) provides a record of all the user's orders. It allows users to track their active orders and review their completed orders. An overview of the order history page is depicted below.
The order history is divided into two sections: Active Orders and Completed Orders.
Active Orders lists all ongoing orders. Each order is represented by an Order Card, which displays the title, status, description, and user. It also provides action buttons that allow the user to interact with the order. The figure below shows examples of these action buttons in use, including the Request Details modal and New Review modal.
Completed Orders provides a list of all completed, cancelled, and rejected orders. It also displays the user's reviews for each order for transparency and accountability.

3.1.4.3 Searching Services

The services page contains category buttons and service cards. Each service card displays the service image, title, username, description, review score, and number of reviews, as well as the price. Clicking on a service card navigates to that specific service page.

3.1.4.4 Viewing Service Reviews

On the service listing page (Fig. 1.4, if you scroll to the bottom, each review includes the username of the reviewer, the score they gave, and a description of their experience with the service.

3.1.4.5 Ordering Services

Fig. 1.8 shows the order input modal, which can be accessed from the service listing page. The user enters a description of the order as well as their location.

3.1.4.6 Providing Additional Details

After placing an order, if the service provider has requested more details, the customer is able to provide these details through a custom modal.

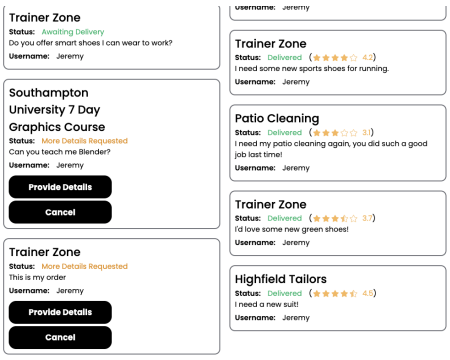


Fig. 3.7. Request more details order history

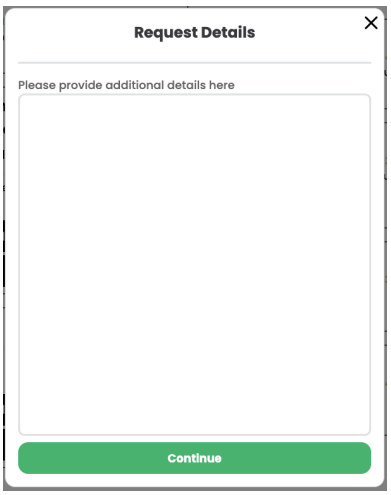


Fig. 3.8. Request more details modal

3.1.4.7 Providing Reviews

After a service provider delivers an order, the customer is able to provide a review for this service through a custom modal.

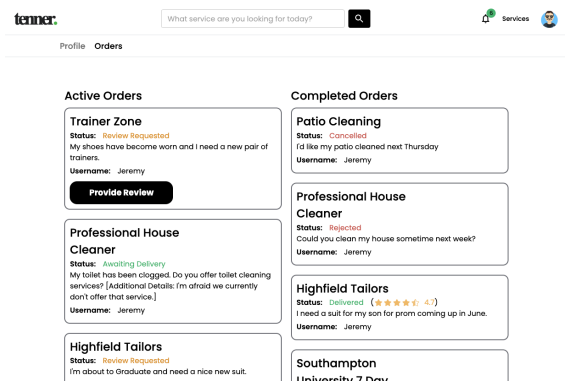


Fig. 3.9. Customer review order history

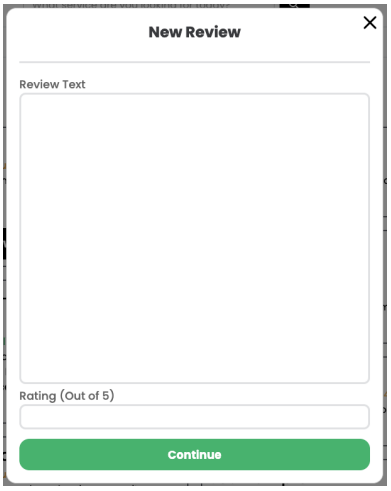


Fig. 3.10. Customer review modal

3.1.4.8 Customer Notifications

When the user has notifications, these appear at the top of their screen in the icon button, which has a badge component to display the number of notifications as seen in Fig. 3.11. The notifications menu is composed of notification items, which can be dismissed or clicked to action them. Actioning them can navigate to a relevant page or open a relevant modal. Different statuses have different colours.

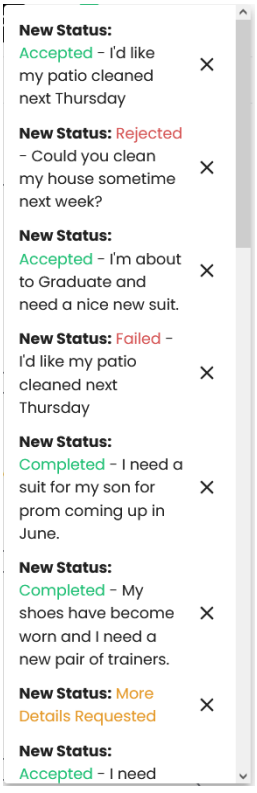


Fig. 3.11. Notifications icon and menu, split in half.

3.2 Advanced Requirements

3.2.1 Social Media Logon

As previously discussed, a user can logon and register with social media login as well as email and password. For the purpose of this project, both Google sign in and Facebook sign in have been implemented. The sign in flow is provided by the Firebase Authentication SDK, where the *GoogleAuthProvider* and *FacebookAuthProvider* can be invoked with the *signInWithPopup* method, as shown in the code snippet below.

Listing 6: Authentication using the *GoogleAuthProvider* and *signInWithPopup* method

```
export const googleSignIn = async () => {
  const provider = new GoogleAuthProvider()
  return await new Promise<UserCredential>((resolve, reject) => {
    signInWithPopup(auth, provider)
      .then(user => {
        resolve(user)
      })
      .catch(err => {
        reject(err)
      })
  })
}
```

As this function from the auth.ts API returns a Promise, containing the UserCredential, the method can be chained with *getAdditionalUserInfo* to identify if the user is a new user. In this case, the application will navigate to the Welcome modal so the user can finish initialising their account.

Listing 7: Handling new users authenticated with social sign in

```
const google = () => {
    googleSignIn()
      .then(result => {
        const additionalInfo = getAdditionalUserInfo(result)
        if (additionalInfo?.isNewUser) {
          setAuthModalState && setAuthModalState('WELCOME')
        } else {
          requestClose()
        }
      })
      .catch(err => {
        if (err.code !== 'auth/popup-closed-by-user') {
          setErrorText && setErrorText(err.message)
        }
      })
}
```

3.2.2 Cloud Hosting

Through Firebase Hosting, Tenner is available as a publicly accessible website at the following URL https://comp6251-465d8.web.app/. Firebase was chosen to host the web application as its services were already used in the application, and thus integration was efficient and simple.

SECTION 4

Testing

4.1 Business Logic

The business logic of our app is tested from two perspectives: do the correct users have access to all and only the functionality they are allowed to access; and if and how does the user interface update when the functionality is completed? The main pages that are tested are the profile page, the service listing page and the header.

4.1.1 Profile Page

User Input	Result	Expected
Edit Profile	Any authenticated customer and provider can edit only their profile.	✓
...	The edit profile modal should disappear when a valid edit is completed.	✓
New Service	Only an authenticated and approved provider can create a new service.	✓
...	The new service modal should disappear when a valid service is created.	✓
Edit Service	Only an authenticated and approved provider can update their own services.	✓
...	The edit service modal should disappear when a valid service is updated.	✓

4.1.2 Orders Page

User Input	Result	Expected
Approve Order	Only providers can approve orders made to their own services.	✓
Reject Order	Only providers can reject orders made to their own services.	✓
Request Details on Order	Only providers can request more details from orders made to their own services.	✓
Provide Details on Order	Only customers can provide more details to orders they place after requested by the provider.	✓
Cancel Order	Providers and customers can both cancel their associated orders. Providers can do this on approved orders and customers can do this when requested more details.	✓
Deliver Order	Only providers can deliver orders made to their own approved services.	✓
Review Order	Only customers can provide reviews to orders they have had delivered to them.	✓

4.1.3 Service Listing Page

User Input	Result	Expected
Placing an Order	Only logged-in customers can place orders for services.	✓
...	The new service request modal should disappear when a valid order is placed.	✓

4.1.4 Header

User Input	Result	Expected
Log In	When no user is signed in, the sign in and join buttons are visible.	✓
...	When a user is logged in, they can see the profile icon button and a logout button in the dropdown.	✓

4.2 Compatibility

The compatibility of Tenner was assessed according to three aspects:

- **Responsiveness**: The responsiveness of the application to different screen sizes was tested using the responsive feature of the developer console within Google Chrome. In this testing, no cosmetic or performance issues were found.
- **Browser Compatibility**: The web application was tested across Google Chrome, Firefox, Safari and Brave and showed no performance or cosmetic issues.
- **Device Compatibility**: The web application was tested on laptop, desktop and mobile devices and showed no performance or cosmetic issues.

4.3 Input Checking

The following modals and search bars all allow for user input to be tested. There is a collection of test cases for each and provided the result compared to what we expected. There is also a record of whether the rest of the user inputs in each modal were saved when submitting an error case.

- Sign Up and Sign In Modals
- Create and Edit Service Modals
- Create Service Request Modal
- Request Details and Review Modals

4.3.1 Authentication

Input	Result	Expected	User Input Retained
No email address	*'Please provide a valid email address'*	✓	✓
Invalid email addresses • testemail.com • test@email.c	*'Invalid email address'*	✓	✓
Valid email address • test@email.com	**Accepted**	✓	✓
No password	*'Please provide a password'*	✓	✓
Invalid password • test	*'Password must be at least 6 characters'*	✓	✓
Mismatching password and confirm password	*'Passwords do not match'*	✓	✓
No confirm password	*'Please confirm you're password'*	✓	✓
Valid password	**Accepted**	✓	✓
No username	*'Enter a username'*	✓	✓
Valid username	**Accepted**	✓	✓

4.3.2 New Service

Input	Result	Expected	User Input Retained
No service title	*'Please provide a service title'*	✓	✓
Valid service title	**Accepted**	✓	✓
No service description	*'Please provide a description'*	✓	✓
Valid service description	**Accepted**	✓	✓
No category	**Accepted**	✓	✓
Valid category	**Accepted**	✓	✓
No availability	*'Please provide an availability'*	✓	✓
No availability times	Availability modal doesn't accept	✓- Should provide an error message too	✓
Valid availability	**Accepted**	✓	✓
No location	*'Please provide a location'*	✓	✓
Valid location	**Accepted**	✓	✓
No image	**Accepted** - Uses default image	✓	✓
Valid image	**Accepted**	✓	✓
No price	*'Please provide a price'*	✓	✓

Invalid price • test • 3 • 3.333	Users can only type numbers and modal will automatically create valid price • *Doesn't fill in box* • £3 • £3.33	✓	✓
Valid price	**Accepted**	✓	✓

4.3.3 New Service Request

Input	Result	Expected	User Input Retained
No service request description	*'Please provide a description'*	✓	✓
Valid service request description	**Accepted**	✓	✓
No location	*'Please provide a location'*	✓	✓
Valid location	**Accepted**	✓	✓

4.3.4 Orders Modals

Input	Result	Expected	User Input Retained
No request details description	*'Please provide more details'*	✓	✓
Valid request details description	**Accepted**	✓	✓
No review description	*'Please provide a review'*	✓	✓
Valid review description	**Accepted**	✓	✓
No review rating	*'Please provide a rating'*	✓	✓
Valid review rating	**Accepted**	✓	✓

4.4 Firebase Rules

Firebase real-time database rules are put in place to restrict what calls can and cannot be made.

- Service requests can only be seen by the authenticated customer or provider linked to it and only created by authenticated customers.
- Services can be seen by anyone and can only be created by authenticated & authenticated providers.
- Reviews can be seen by anyone and can only be created by authenticated customers for their own service requests.
- Users can be seen by anyone and created by anyone.

These rules are reflected in the programmed set of rules in our Firebase as shown below.

Listing 8: Firebase Rules

```
{
  "rules": {
    "ServiceRequests": {
      "$serviceRequestID": {
        ".read": "auth␣!==␣null␣&&␣(root.child('ServiceRequests').child($serviceRequestID)↩
            ↪.child('providerID').val()␣===␣auth.id␣||␣root.child('ServiceRequests').child↩
            ↪($serviceRequestID).child('customerID').val()␣===␣auth.id)",
        ".write": "auth␣!==␣null"
      }
    },
    "Services": {
      ".read": true,
```

```
      ".write": "auth␣!==␣null␣&&␣root.child('Users').child('Provider').child(auth.uid).↩
        ↪child('status').val()␣===␣'Approved'"
    },
    "ServiceReviews": {
      ".read": true,
      ".write": "auth␣!==␣null␣&&␣root.child('Users').child('Provider').child(auth.uid).↩
        ↪child('status').val()␣===␣'Approved'"
    },
    "Users": {
      "$uid": {
        ".read": true,
        ".write": true
      }
    }
  }
}
```