**Design, Analysis and Implementation Decisions for Make Waves Recording Studios**

**Initial Design Thoughts**

For Make Waves, the idea was to build a slick, responsive web app for a modern recording studio — something that looked creative but still professional. I wanted the design to feel bold and stylish, with a strong focus on usability. The layout is mobile-first and the whole experience is designed to feel smooth and polished on any device. The design evolved with ongoing experimentation. As you can see, I began the process with a brainstorm after a few logo design changes.

**Home Page**

The landing page makes an impact with a clean background video, a strong logo, and clear calls to action. It’s built to adapt smoothly on mobiles, tablets, and desktops — so it looks great whether you’re browsing on your phone or a full-sized screen.

**Services Page**

This page breaks down everything the studio offers. Each section is well-spaced and easy to skim, with a clear visual structure that helps guide users through the content. Everything’s optimised to look tidy and readable, especially on smaller screens using cards.

**About Page**

This tells the story behind Make Waves. I’ve layered text and images so it feels engaging without being cluttered. It’s styled to keep the focus on the message — who we are and what we’re about whilst implementing a clean slider gallery.

**Login & Registration**

The login and sign-up forms use Flask-WTF for security and validation with a clean design implementing bootstrap styles.

**Booking Page**

Once you’re logged in, you can book a studio session using a clean form built with Flask-WTF. Everything’s protected with CSRF and there’s built-in validation to keep things secure.

**User Dashboard**

Once logged in, the user has a view of all of their details, upcoming bookings, options to edit/delete their bookings and an edit profile option.

**Edit Profile**

From here, users can update their personal details for their profile. I have kept this page simple for ease of access.

**Admin Dashboard**

Admins get access to a private dashboard where they can view and manage all user’s bookings. It’s basic but functional — enough to support real use cases without being over-complicated, given the size of the web application.

**Custom Error Pages (403, 404, 500)**

Instead of boring default error pages, I’ve added fully styled versions that match the rest of the app. Navigation is hidden to keep things clear, and users get simple routes back to the home page or login.

**Main design features**

The Make Waves app was built using Flask as the core framework, combined with Jinja templates for dynamic rendering of HTML pages. Flask made it straightforward to map routes, manage views, and structure the backend cleanly while keeping things lightweight and flexible.

For handling forms, I used Flask-WTF, which provides CSRF protection and built-in validators. This was essential for secure login, registration, and booking features — it helped keep forms tidy, reliable, and resistant to common web vulnerabilities.

Styling was done using Bootstrap 5 to create a consistent and responsive layout. While Bootstrap handled the foundations, I added custom CSS and media queries to fine-tune how the site looked across mobile, tablet, and desktop. The site adapts cleanly across screen sizes, with specific logic to switch logos and navigation styles based on viewport width.

To keep feedback non-intrusive, I implemented toast notifications. These pop up in the top corner of the screen with real-time success or error messages — useful for actions like submitting a booking or logging in.

On the backend, I used pgAdmin4 with PostgreSQL to manage the database. This handles all structured data like bookings, users, and admin roles. SQLAlchemy works with Flask to make database interactions simpler, while migrations are managed with Flask-Migrate for smooth schema changes. Render hosts the database, alongside the app.

The app is deployed live via Render.com, with environment variables used for managing sensitive data like secret keys and database URIs. This setup keeps the deployment secure and production-ready, with minimal manual config needed during each update.

Finally, for optimised image delivery, I’ve integrated Cloudinary. This makes sure user’s uploaded profile images are compressed, cached, and delivered fast — which helps both performance and user experience, especially on slower mobile connections.

**My Development Process**

I started by sketching out some rough wireframes and designing the logo on Canva — just to get a feel for the brand direction. Once I was happy with the overall look, I moved on to building the layout using HTML, CSS, and Jinja templates. I added Flask routes shortly after, then got the forms and user sessions up and running.

A big priority for me was making sure the interface felt clean and professional across all screen sizes. I wrote quite a few media queries to help the layout adjust smoothly on different devices. The logo switches between desktop and mobile versions depending on screen width, and the main nav turns into a hamburger menu on smaller screens — all handled with CSS.

I kept testing everything on both desktop and mobile, ironing out layout bugs and fine-tuning the styles as I went. I also ran Lighthouse audits to pick up any performance or accessibility issues — that helped me catch things like oversized images and unused styles. Switching to .webp images and streamlining the CSS made a noticeable difference.

**Challenges faced during development**

One of the trickiest parts of the project was trying to move my local PostgreSQL database to work with Render’s cloud setup. During development, I was using pgAdmin 4 to manage my database locally, and everything was running smoothly. But once it came to deploying the app, I had to rethink how the database was hosted and connected.

Render doesn’t support hosting a database directly through pgAdmin like I had locally, so I needed to provision a hosted PostgreSQL instance through Render’s dashboard and then update my environment variables to point my Flask app to the correct remote URL.

The database connection string format was different from my local setup, which caused some initial issues with authentication and migrations. I had to carefully update my SQLALCHEMY\_DATABASE\_URI in the .env file, and make sure I was running flask db upgrade correctly so that all the tables were set up properly on the remote side.

Another small but important challenge was making sure that the DATABASE\_URL environment variable was recognised during deployment. Since this didn’t behave exactly the same as my .env in local development, I had to define it directly in the Render dashboard. Once this was set up properly and my tables were migrated, everything clicked into place.

It took a few trial-and-error deployments, but this process gave me a better understanding of how cloud environments differ from local ones — especially in terms of security, connection handling, and config setup.