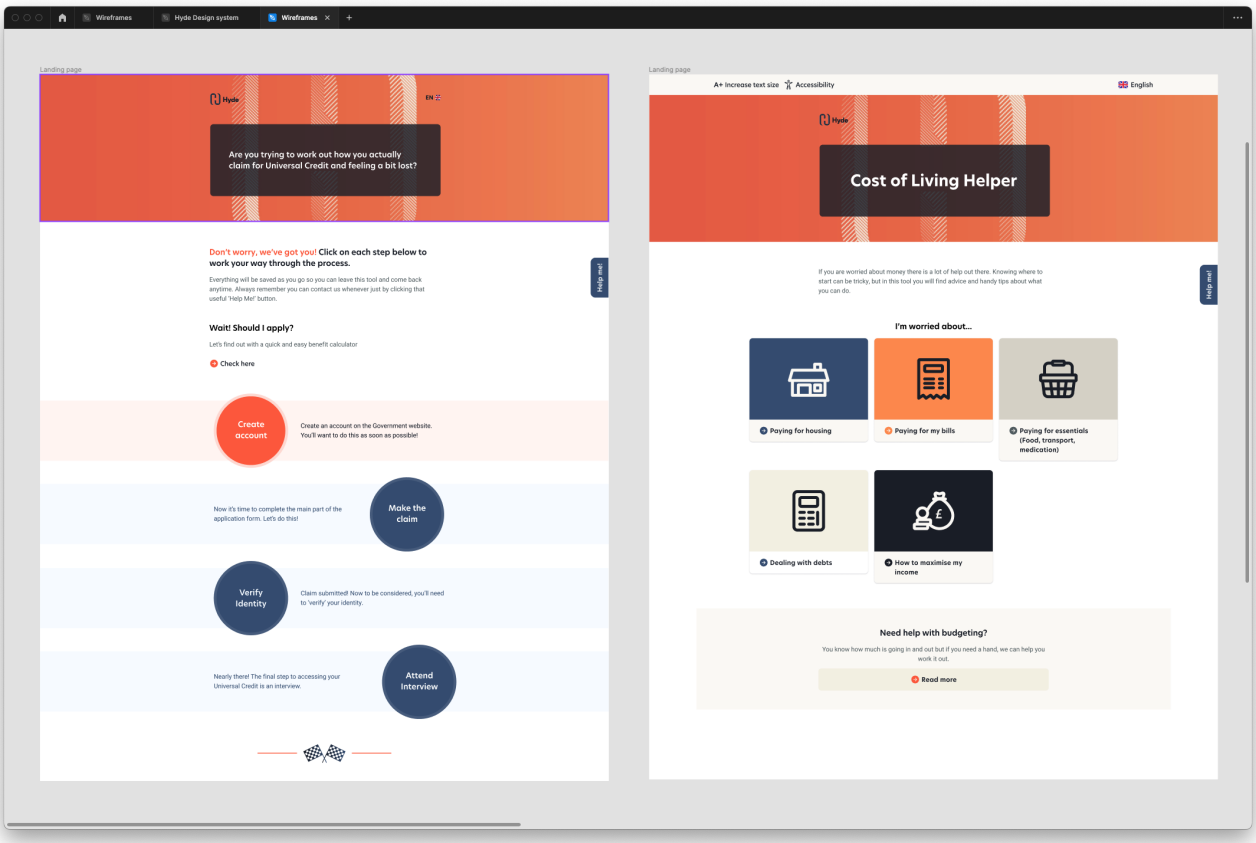
1 - Introduction

The aim of this project was to develop a new web-based application, the Cost of Living Helper (CoL-Helper) for housing provider Hyde. It aimed to assist people affected by the cost of living crisis by collating tools and information into a single resource that was easily accessible and understandable by everyone. The tool was based upon the Universal Credit Helper (UC-Helper) developed by Yalla in 2021 which helped people applying for universal credit.

2 - Scope

The project was a fork of the UC-Helper's codebase, with the aim of adding multi-lingual capabilities, improving accessibility, and a more extensive CMS to both the UC-Helper and the new CoL-Helper. My focus was translating the site's content and adjusting the formatting to suit the reading direction; expanding the reach of the site to people who may not speak or read English as a first language.



↑ Wireframe of UC-Helper and CoL-Helper

There were two key metrics of success for this project, technical and engagement. Technically the project was successful if the code I wrote passed review and testing, was merged into staging and passed quality assurance. Engagement metrics were measured using heat-mapping and page analytics and only became apparent after the product was deployed. The technical metrics fell within the seven week time-frame of this project and were included in the report, however, engagement metrics did not.

Funding for the CoL-Helper was secured on the basis that language and accessibility tools would be added to both projects. The project was completed when these features were signed off by the client.

3 - Plan

3.1 - Overview

I worked through five stages of the eight stage software development life-cycle, research, planning, development, testing and deployment. Design, content creation and maintenance stages fell outside of the allocated seven weeks, and are not therefore documented.

Week 1

- Technical research
- Technical review
- Sprint planning
- Issue and acceptance criteria

Week 2 - 5

- Development sprint

Week 6

- Product demo
- Client quality Assurance
- Bug fixing

Week 7

- Deployment

Week 8 - 9

- Project report

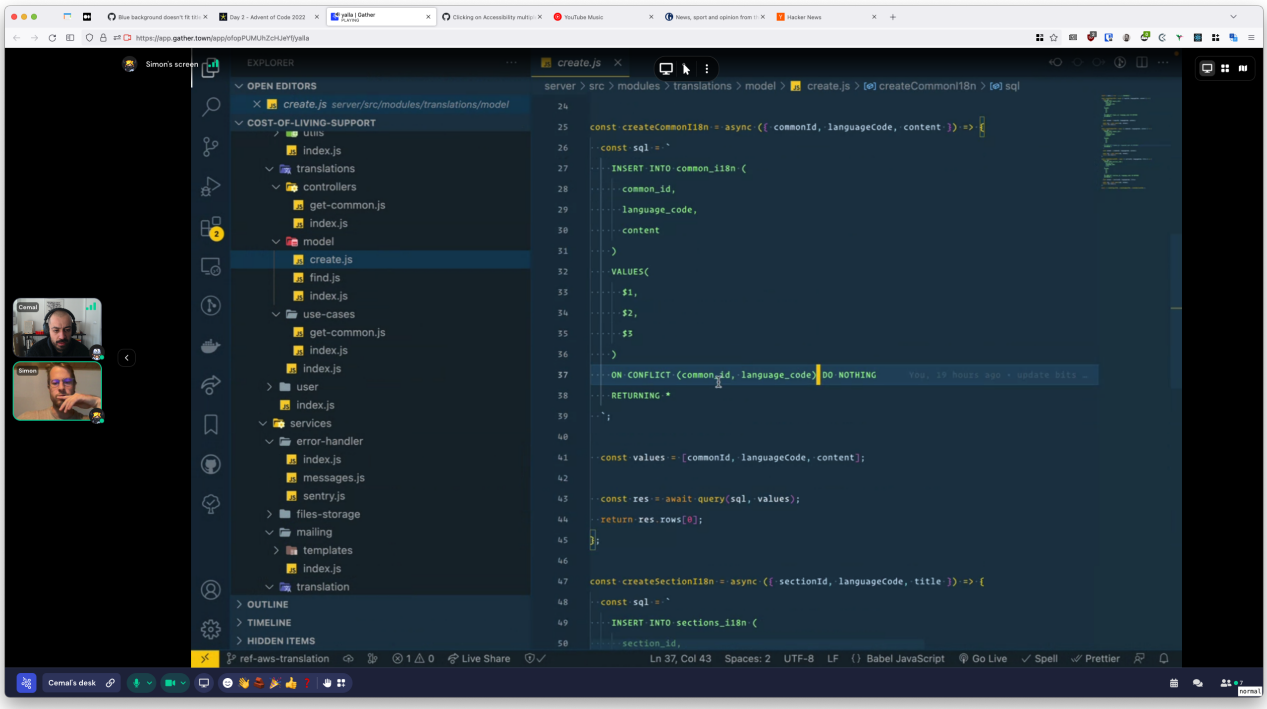
3.1.1 - Roles

There were several roles and responsibilities within the development lifecycle, and there was often overlap between them:

- **Product owner:** Maggie Houghton from Hyde Housing was responsible for representing the client's interests and ensuring that the project met their business goals
- **Project manager:** Erica Gasparini was responsible for managing the overall project, timelines, budgets, resources and acted as liaison between development team and Hyde

- **UX/UI designer:** Jem Abulhawa and Beth Scott were both responsible for conducting initial user testing and creating overall look, feel, user experience, and user interface of the website
- **Lead developer:** Ramy Al Shurafa was responsible for leading the development team and ensuring that the project met requirements and was completed on time
- **Developers:** Fadi Omar, Simon Dupreesi and myself were responsible for coding the website, working closely with lead developer, UX/UI designers and project manager

To ensure that each team member, including myself, was able to make a contribution, we encouraged open and transparent communication, provided necessary resources and support, practiced active feedback through retrospectives, and used techniques such as pair programming and code reviews to encourage teamwork and learning.



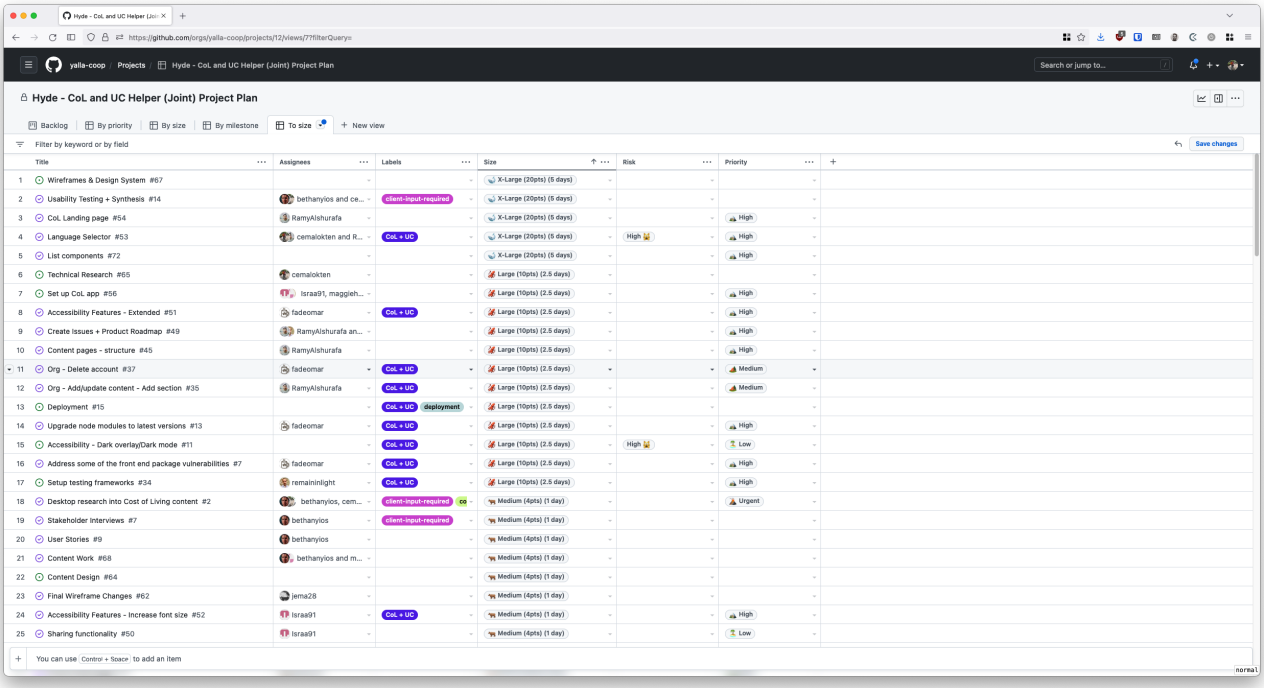
`↑ Pair programming with Simon Dupreesi`

3.2 - Research

The first stage of this seven week project was to research the possible approaches to language translation or internationalisation as it is often known.

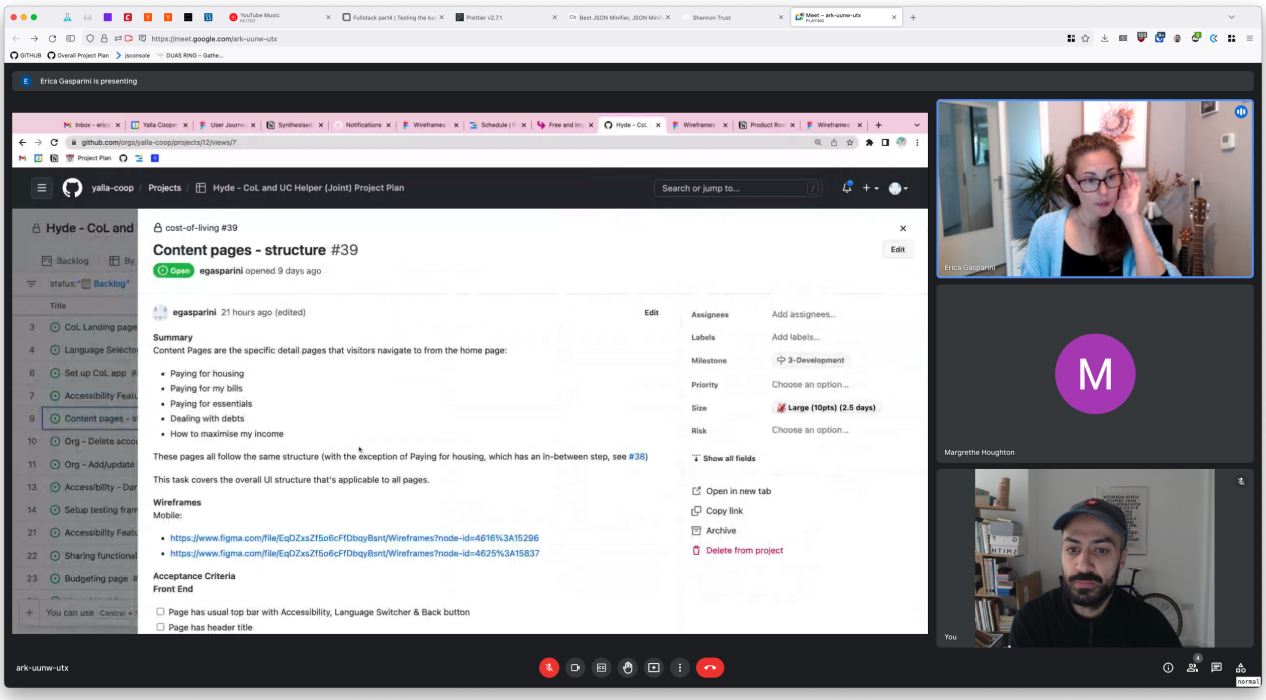
3.3 - Planning

I used Scrum, an Agile framework, to plan and manage the project. It allowed for flexibility, rapid iteration, and regular communication. In contrast, Waterfall is a traditional linear approach where change is discouraged and requirements are set in stone. I planned the development sprint with my project manager and lead developer. Using Scrum, I was able to plan and manage the project flexibly, breaking down user journeys into stories with acceptance criteria on GitHub (see Figure 1, 2, & 3) and updating progress regularly, which helped deliver the project efficiently and effectively.



↑ GitHub project board

We then estimated the amount of time we expected each issue to take, based on previous sprints using a similar tech stack and features. We then added a priority rating depending on its importance. Finally, we assigned a level of risk to each issue based on our experience, and confidence in that task. Language translation was assigned a high-risk factor due to our limited understanding and experience.



↑ Sprint planning

3.4 - Development

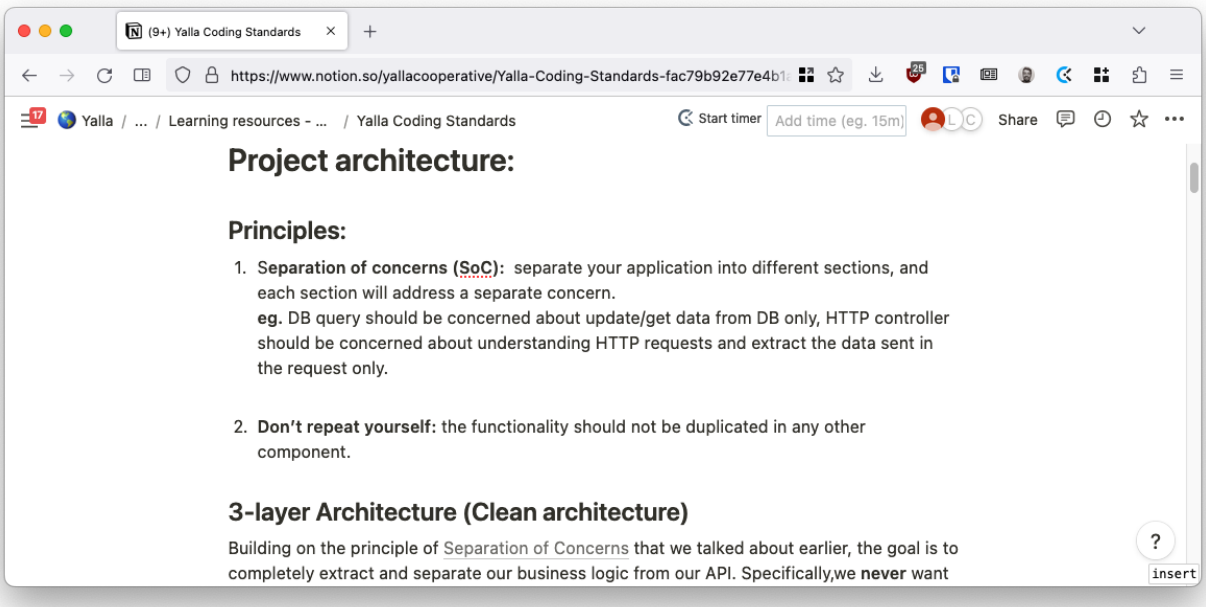
I aimed to complete the front-end work within the first week of the development sprint, with a further two weeks to complete the back-end work, using the remaining week as contingency. After speaking with my lead developer, we decided upon a stack, see Figure 4.

I used a functional paradigm in this project as it was easier for me to understand, modularise, test and debug compared to object-oriented. The functional approach uses functions as building blocks while object-oriented focuses on the use of objects. Object-oriented programming often leads to complex and hard to maintain codebases with too many class dependencies. The functional paradigm allows for easy creation of small, reusable and testable functions.

3.4.1 - Organisational standards

I followed the software design pattern of Separation of Concerns (SoC), which separates an application into distinct sections, each addressing a separate concern. This creates an organised system that can adapt to change. Applying this principle to the language translation feature, I created individual components for the front-end, for example criteria Figure 1.1 and Figure 1.2. On the back-end, I applied the same principle, with logic for criteria Figure 3.2 and Figure 3.3 both stored in different functions, which also made it easier to test and debug.

Where possible, I kept my code DRY (**d**on't **r**epeat **y**ourself), which relates well to SoC, keeping functionality separate means I could export and reuse it, instead of duplicating it. I organised the front and back-end inline with my organisations standards, see Figure 5 and 6.



↑ Coding standards document

3.4.2 - Code style

I used ESLint and Prettier to ensure uniformity, maintainability and code style remained consistent across all developers work. Together, they make the code more readable, consistent and maintainable, making it easier for my team to understand and collaborate on the codebase.

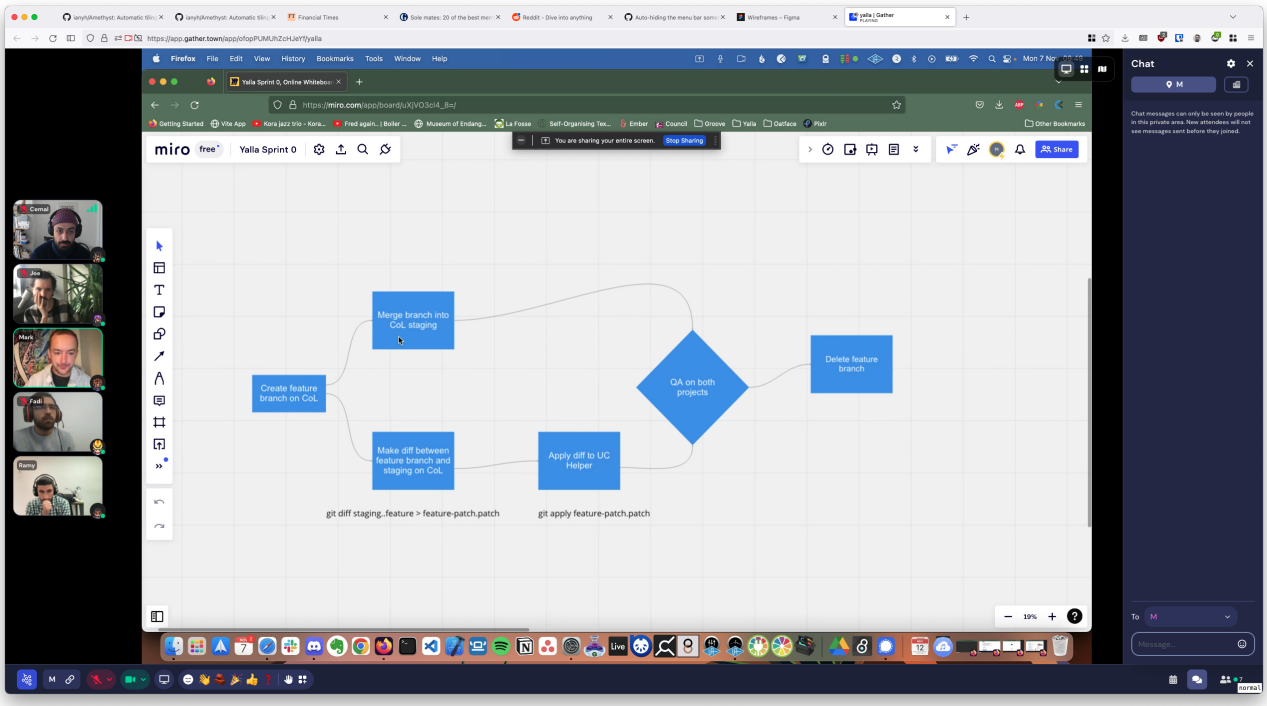
3.4.3 - Source control

While working asynchronously with my team I:

- Committed code with detailed commit messages
- Used branches for new features and bug fixes
- Ensured branches were based on the latest code
- Opened PR's and notified team
- Made suggested changes & requested re-reviews
- Never merged my own code, always done by reviewer

The project scope and funding required that I added the language translation features to the codebase of the UC-Helper as well as to the new CoL-Helper. I followed a workflow created by my lead developer to push code from from CoL-Helper to the UC-Helper:

1. Obtained commit hash of common ancestor of feature-branch and develop-branch (Figure 7)
2. Created a patch file of feature-branch differences (Figure 8)
3. Copied patch file to UC-Helper project, applied to feature branch, verified functionality and pushed changes to remote repository (Figure 9)



↑ Git workflow meeting

3.4.4 - GDPR and cookies

I implemented a cookie banner as required under GDPR, to obtain user consent for the use of analytical technology that stored cookies containing usage data on the CoL and UC-Helper. The banner explained the use of cookies and required users to accept or decline their usage.

3.4.5 - Security

I kept my computer's software updated and used two-factor authentication when available. To secure API keys and secrets for services like AWS Translate, I imported them securely through `dotenv` and added them to my `.gitignore` file.

3.5 - Testing

I conducted tests during and after development, including Storybook for UI components, Playwright for end-to-end tests, and Jest for back-end unit tests. I also collaborated with the client to conduct user acceptance tests.

3.6 - Deployment

I deployed the CoL-Helper and updated the UC-Helper to a Heroku staging server, the same environment as production, to minimise issues when deploying to production. I selected Heroku because it eliminated obstacles such as server configuration and database setup with minimal configuration.

The original UC-Helper did not use containers and instead used a standard local development environment. Due to time and budget constraints, we kept the local development environment and did not containerise the application.

4 - Research

4.1 - Technical spike overview

I conducted a technical spike into language translation, driven by a lack of understanding from my team and myself, as well as the client's need to keep costs low with a small budget for ongoing maintenance and development.

I researched five popular translation services based on language availability and cost, all of which required sending page content to an external server for translation and computation. Each service charged per character, including spaces. I also researched four popular frameworks for implementing translations on the page, which provided a way to insert translations, fallbacks, and detect browser language. I then made suggestions for three different approaches:

Option 1: Dynamic translation with no storage

- Text translated on demand with no local or remote storage

Option 2: Remote storage of translations generated during development

- Text translated once manually and stored in a database

Option 3: Dynamic translation with local storage

- Text translated once and stored programmatically
- Translations retrieved from database to reduce requests and lower costs
- Translations cleared from database when content is updated to ensure they remain up-to-date

4.2 - Technical recommendations

I concluded that option 1 would be the easiest to implement but the most costly, option 2 would require more ongoing development time but had the lowest up-front cost, and option 3 was the most complex to implement but had the benefits of both option 1 and 2, requiring no developer interaction to add or update translations while also having the cost and speed benefits of translating a text only once and retrieving it from the database. However, option 3 would require the most upfront development time.

I recommended option 3, with a combination of i18next internationalisation framework and AWS Translate as the client already had services running from AWS. In summary:

- Reduced costs by only translating content once and storing it in a database
- Increased speed by retrieving translations from the database after the initial translation
- A large selection of languages

- No need to manually translate and store the data, it could be done on-the-fly using the entire selection of languages from AWS
- i18next framework, which is well-maintained, has extensive documentation, and supports React

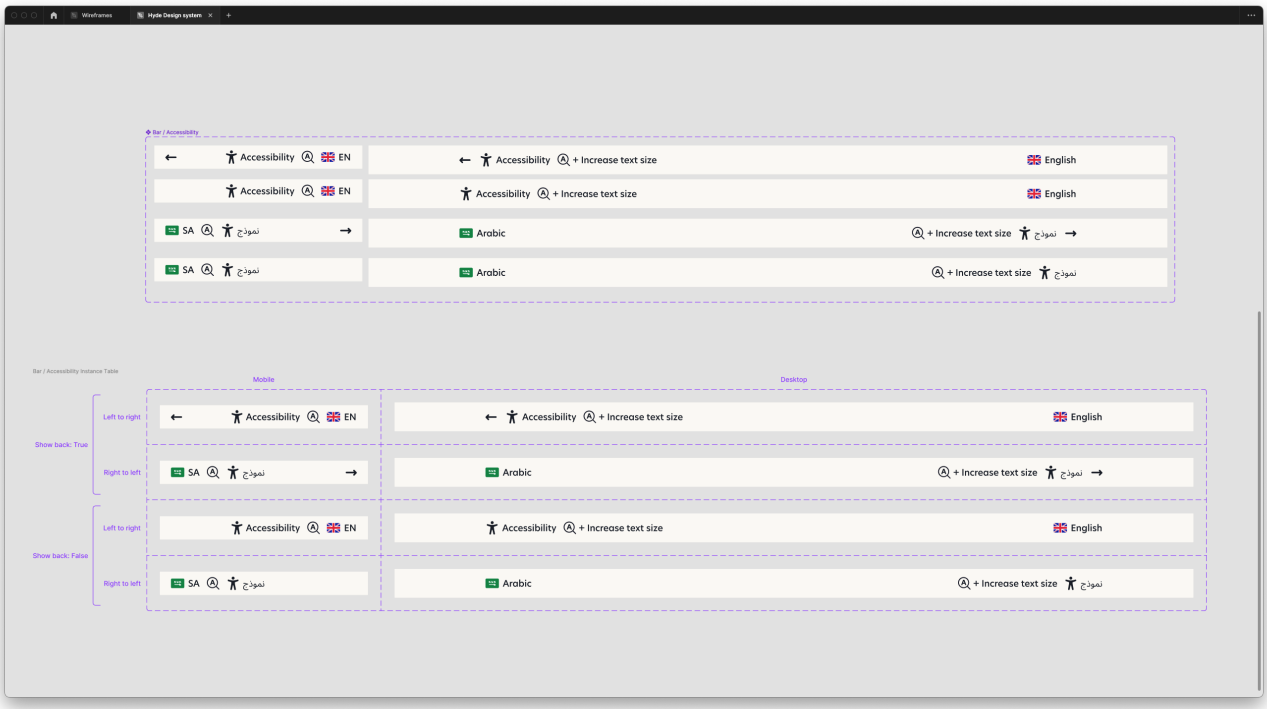
5 - Outcome

5.1 - Front-end

5.1.1 - Language bar

First I worked on the front-end components, criteria Figure 1.1 - 1.6. The file structure and naming convention I used can be seen in Figure 10.

I started with the `LanguageBar` component, with four variants (left-to-right desktop, left-to-right mobile, right-to-left desktop, and right-to-left mobile), it sat at the top of every page. Left-to-right variants were used for languages like English while right-to-left variants were used for languages like Arabic, as per criteria Figure 1.7.

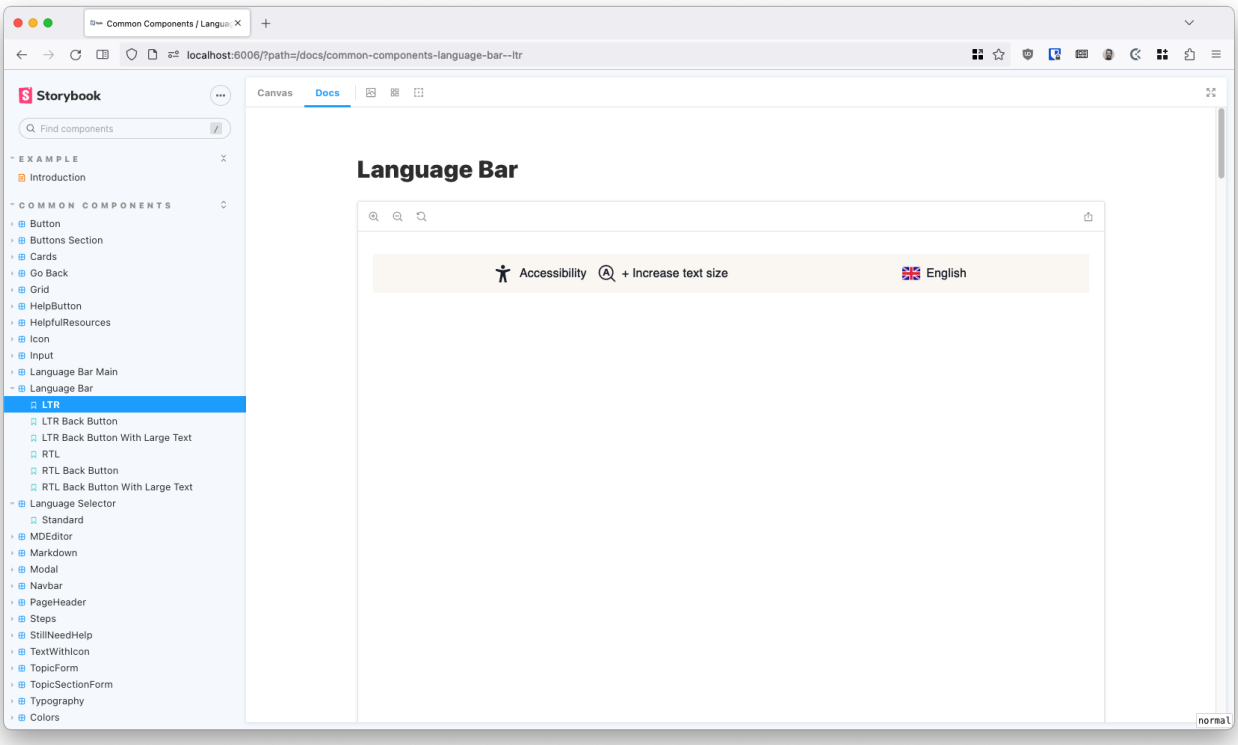


`↑ Language bar wireframes`

I created separate functions for desktop and mobile versions, each with two return values based on the results of the internal functions. I exported a single function called `LanguageBar`, which would output one of four variants depending on the input (see Figure 11).

I used two functions from external packages, one from `i18next` which returned the language direction based upon the selected language, for example `en` would return `ltr`, I could use this in a ternary statement to return the correct component. The second function I used was from `react-responsive` which included `useMediaQuery` that returned true if the conditions for the breakpoint were met (see Figure 12). The `LanguageBar` component code looked approximately like Figure 13.

I used Storybook to test the component in isolation. This allowed me to focus on the specific component and its behaviour, without the need for a full application setup. I was able to render the component, examine its output and interact with it in various states. This helped me to identify and fix issues with the component, and ensure that it met the design and functional requirements. Additionally, by using Storybook, I was able to demonstrate it to other members of the team, allowing them to see how it works and how it should be used in the application.

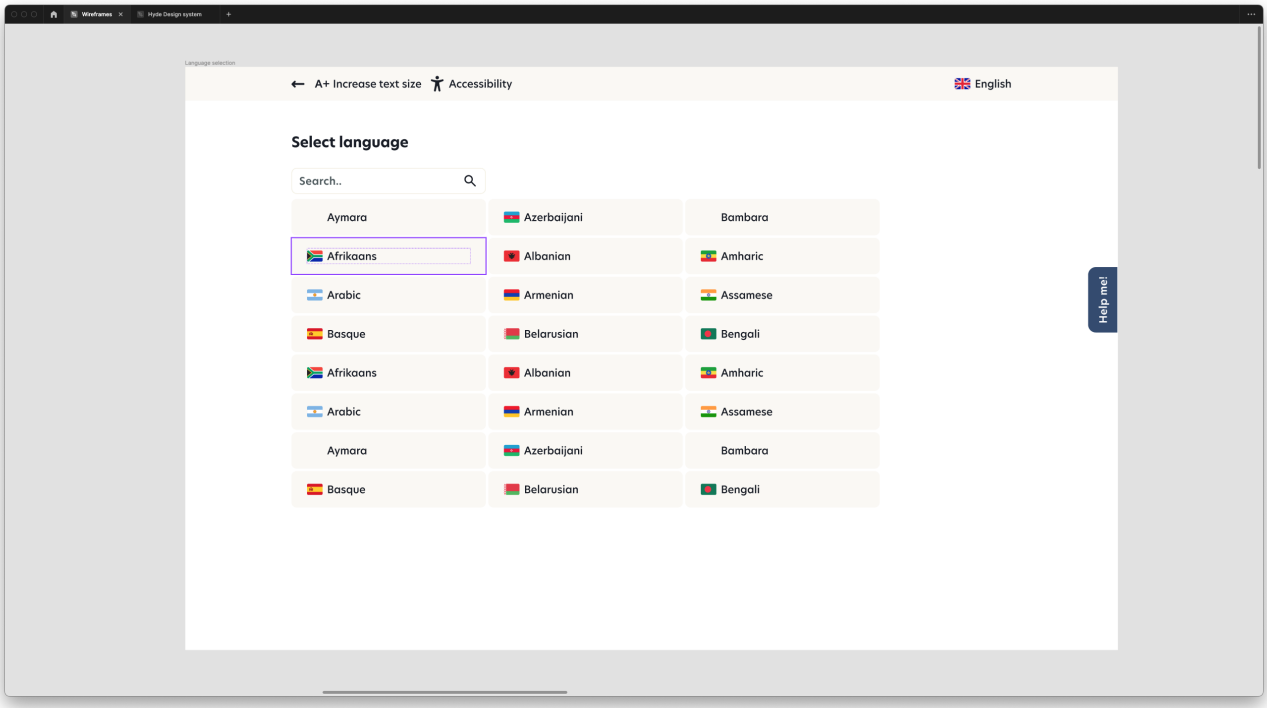


`↑ Testing language bar with Storybook`

I styled the components using Emotion, a library for writing CSS styles with JavaScript; it allowed me to define styles within the component code, making it more modular and maintainable, as well as dynamic and responsive based on component state or props. Each component had its own `style.js` file as per Yalla's organisational standards, where I defined styles, passed in props for dynamic changes, and imported constant variables (see Figure 14).

5.1.2 - Language selector

The language selector is a full page menu of languages to translate into, each with a flag and a search box to filter languages by their full name or language code.



`↑ Language selector wireframe`

I created the component and styled them in the same manner as the `LanguageBar` component. I used `display:grid` to create a column based layout, using a media query to adjust the layout depending in the width of the page as the `LanguageBar`.

To add languages to the page, I created an object inside `constants/data-types.js`, containing the language name and the language code. I stored this object inside the constants folder as it would need to be reused (see Figure 15)

`data-types` was imported as a single object, to access the `languageCodes` object I used dot notation, and passed this variable into `Object.entries` which would convert the key value pair into an array of arrays (see Figure 16). Using the JavaScript `map()` method I was able to extract the language and the language code into separate components (see Figure 17).

I implemented the search feature using the React `useState` hook and the `filter()` method. The text input component `BasicInput`, called a passed-in function with the input value, updating a piece of state. I used the state value in a `filter()` method called on the `languages` array, updating the results and re-rendering the page as the user inputs a search term (see Figure 18). The search returned items matching the full language name or the language code, as per requirements Figure 1.5 & Figure 1.6.

5.1.3 - Front-end i18n

I then worked on implementing `i18next` on the front-end for common words, such as buttons, introductions and titles. Implementing it for a small set of words and only on the front-end helped me to gain experience with the `i18next` framework before moving onto the more complicated implementation on the back-end.

I read through the `i18next` documentation to get an overview of how to set it up and it's capabilities. To install `i18next` I used NPM (node package manager). Here is how I set up and started using 18next:

1 - Install the i18next package:

```shell
npm install i18next
```

2 - Import the library into a JavaScript file:

```javascript
import i18next from 'i18next';
```

3 - Initialise i18next and set the language to `en` and add a key called `hello`:

```javascript
i18next.init({
    lng: 'en',
    resources: {
        en: {
            translation: {
                "hello": "Hello World"
            }
        },
        fr: {
            translation: {
                "hello": "Bonjour le monde"
            }
        }
    }
});
```

4 - Use the `t` function provided by i18next to translate a key:

```javascript
i18next.t('hello'); // Output: Hello World
```

5 - To change the language, I used the `changeLanguage` method, this will search the resources object for `fr` and if it exists display the relevant keys. `i18next` stores the

selected language in the users `localstorage` so a user can select a language and have it persist between visits:

```JavaScript
i18next.changeLanguage('fr');

i18next.t('hello'); // Output: Bonjour le monde
```

6 - Separate languages out into their own `JSON` files:

```JavaScript
i18next.init({
    lng: 'en',
    resources: {
        en: {
            translation: require('./locales/en/translation.JSON')
        },
        fr: {
            translation: require('./locales/fr/translation.JSON')
        }
    }
});
```

7 - To implement translations in React use the `t` function:

```JSX
import { useTranslation } from 'react-i18next';

function MyComponent() {
  const { t } = useTranslation();
  return <h1>{t('hello')}</h1>
}
```

If a key does not exist for a particular language, it will fallback to the configured fallback language, e.g `en`, these settings are specified inside the `init()`. I created an `i18n.js` in the root of the client folder, this contained all the settings for `i18next` which I then imported into the entry point of the client, `index.js` and used throughout the front-end (see Figure 19).

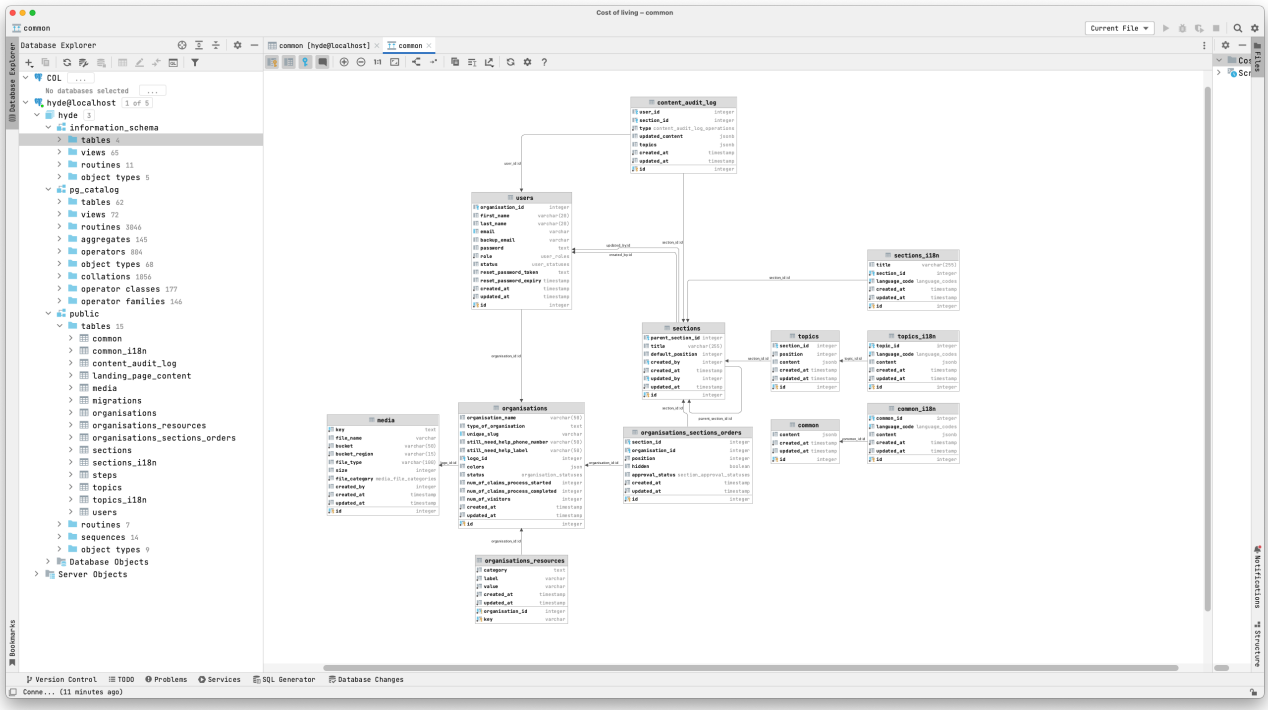
I organised translations using `i18next`'s concept of namespaces, which are sets of related texts that can be loaded as needed instead of loading all texts for the entire site. I used the 'common' namespace to store phrases, words, and paragraphs used throughout the site. This made it easier to manage and maintain translations over time.

5.2 - Back-end

The CoL-Helper and UC-Helper's content were both stored in a Postgres database accessed when the user visited the site. A relational SQL database is a better solution for large sites with interconnected data and frequent updates than using a JSON file because it scales better, allows concurrent sessions and enforces data integrity. My next task was to use the Postgres database to store translations instead of a JSON file.

The back-end implementation entailed retrieving content from the database and sending it to the front-end, parsed into `i18next` as a namespace, displaying the translated value. If a selected language's translation was not in the database, it sent text to the language API, translated and stored in the database with the language code as a field. Future requests for that language were fetched from the database, advantages of this approach include:

- Keeping client-side code clean and maintainable
- Increased efficiency by storing translated text in the database and fetching it for future requests, avoiding unnecessary API calls
- Fallback to English for missing translations in the database to ensure no missing translations for users
- Properly organising the database by storing translated text with the appropriate language code as a field



`↑ Database schema for CoL-Helper`

I updated the database schema with my lead developer to ensure compatibility with existing data. A well-designed schema is crucial for clear and consistent data storage, better understanding of data relationships, effective querying, data integrity, performance, scalability, and team collaboration.

I decided to keep the English content in a separate table from the translations in the database. This allows for easy management and updates of the source content without affecting the translations. Moreover, this separation makes it simple for me to add new languages without the need to modify the existing structure of the database and to identifying missing translations. All in all, having two tables for source and translations is a more organised and efficient approach to localisation.

I started by adding the common words to the build steps of the database, whenever the database is being built, it would add the common words to the common table. I did this by writing a function for adding content to the database programatically (see Figure 20).

I followed clean architecture principles (route → controller → use-case → query) to create an endpoint for accessing data on the front-end. Clean Architecture, is a software design principle that separates an application's concerns into separate layers, this helped me and my team understand and maintain the application better. I wrote a unit test to confirm the route for fetching common translations returned the expected output (see Figure 21).

I then updated the `getCommon` SQL query to take a language code as an argument, and return the `common` object for that language if it exists, otherwise return English. I did this by using `COALESCE`, I wrote and tested this query using Beekeeper Studio (see Figure 22).

The `use-cases/get-common.js` function used business logic to check if a translation existed in the database for the specified language code. If it didn't, it sent the translation request to the API and stored the response in the database. I put this function in `use-cases` because it translated content based on a set of rules. If the translation depended on input from the client, it would have been placed in `controllers`. This function fetches the common words from the database, passing the language code as an argument (`lng`), `Translation.getCommon()` will return the common object with the matching language code or if it does not exist, return the content object matching the `en` language code (see Figure 23).

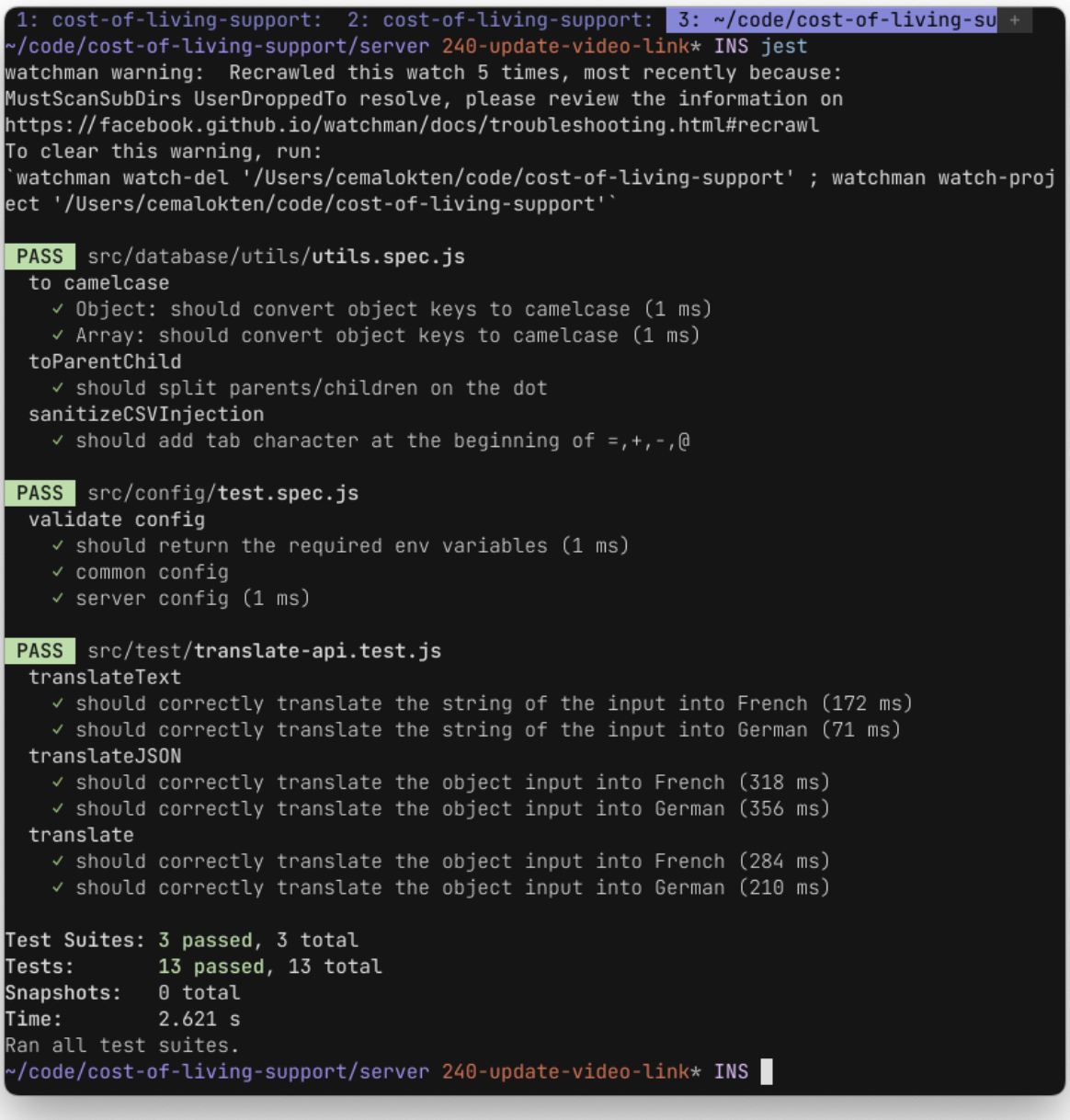
The next part of the function is `translateContent()`, it takes two argument's, `lng` and `contentArray`. It checks if the response from the database matches the language code provided, if it does then it simply returns an object with the common words; if the response language code does not match the one provided it sends the common words to the translation API, and returns a translation.

After the text has been fetched, checked and potentially translated it is passed to `Translation.createCommonI18n` which loops through the `commonT` array, if the `isTranslated` key is falsey, it adds it to the database with the correct language code.

The `getCommon()` function is an algorithm that takes in an input of `lng` and returns translated common words based on the specified language. It follows a set of instructions such as fetching common words, translating them, checking if their already translated, and creating new translations if needed to achieve the goal of providing translated common words. These steps are executed in a specific order and can be considered an algorithm.

5.2.1 - Translation logic

I wrote a series of unit tests to validate the three functions that made up the Translation algorithm (see Figure 24). These functions perform the single task of translating an object of key-value pairs, leaving the keys unmodified for referencing in the code while translating the values. These tests and their output helped to efficiently identify issues while I was writing the functions, rather than having to manually test each individual function as I went.



`↑ Unit test results`

The `translate` function at the bottom of Figure 25 requires four parameters: `source`, `target`, `JSON`, `id`. Source is the original language of the text, target is the language to translate into, JSON is the text to translate, and the id is an identifier. The function first checks all the required parameters are provided, if not it throws an error.

`translateJSON()` takes an object, target language and source language. It iterates through the object, checking if the value at each key is also an object, if it is, it recursively calls the translate JSON function passing the object value and the same parameters; else, it uses the `translateText` function to translate the text. If the translation is successful, it returns an object containing the passed in parameters and the translated content, if unsuccessful it throws an error.

I created a helper function that contained general translation and `i18next` functions that I used many times throughout the app, I imported this helper function and destructed the functions within it that I required (see Figure 26).

5.3 - Linking front and back-end

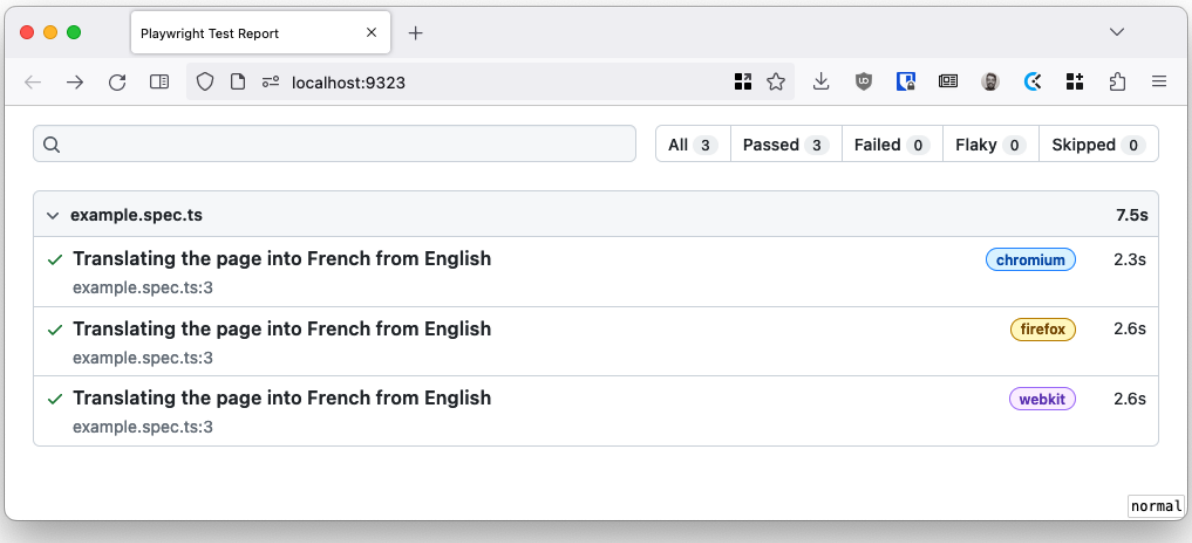
I linked the front and back-end using AXIOS, a JavaScript library for making HTTP requests. It is widely supported, well-maintained and offers a wider range of browser support than the built-in `fetch()` function. Since the users of the CoL-Helper will be using various devices and browsers to access the site, it's important to support as many of them as possible.

All API calls on the front-end are stored in the `src/api-calls` folder. The `getCommon` function makes a HTTP request to the `/translations` route defined in the back-end, sending the language code as a parameter. It then returns the data or throws an error (see Figure 27).

To make the common translations available throughout the front-end, I created a context in React. I fetched the common data once and made it accessible to the entire app using a context consumer. Inside the context I fetched the data via the API call inside a `useEffect()`, a built in hook which runs the code inside of it when the component mounts and only again if the arguments passed to it change (see Figure 28).

The `useEffect()` hook fetches the translation for the provided language code and adds it to the common namespace, accessible by the `t()` function, it also stores it in state which is later added to the context provider. When the user selects a language, it calls the `i18next.changeLanguage(lng)` function, updating the state of `lng`, triggering `useEffect()` to fetch a new translation and re-render the page with the updated translation (see Figure 29).

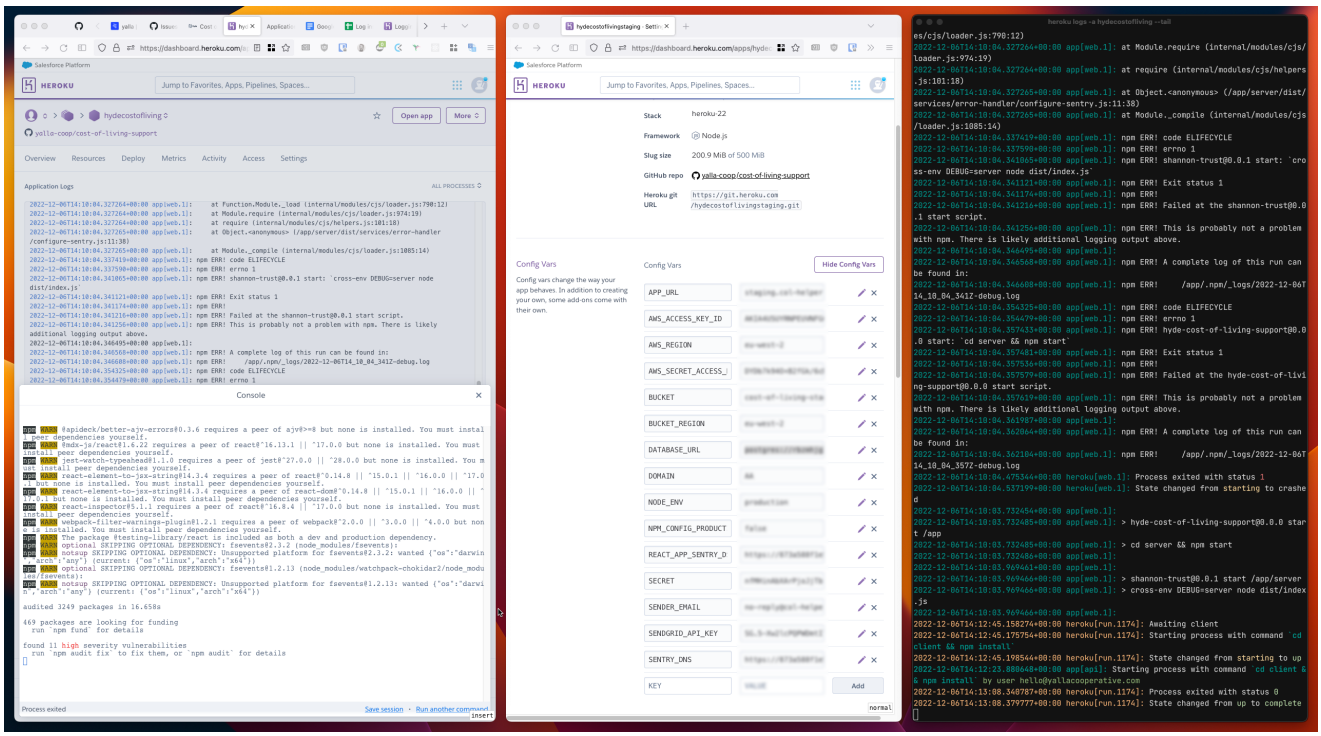
I wrote an integration test using Playwright (Figure 30) to verify translations were properly received by the front-end and rendered on the page. This test simulated a user requesting a translation and checked if the expected translation was displayed.



↑ Integration test results

5.4 - Deployment

I deployed UC-Helper and CoL-Helper on Heroku and used sub-domains of production for testing by client and Yalla. I used Heroku's remote console for debugging and managing dependencies. Deploying to the same environment for both staging and production reduced potential issues from different configurations and dependencies, but also had a risk of both production and staging going down at the same time.



↑ Deploying with Heroku

5.5 - Bugs

I identified issues by using the browser's developer tools and server logs in the terminal to check for errors and bugs by using console print statements and break-points. Additionally, I gathered feedback from the user testing stage and analysed unit and end-to-end test results to identify areas for improvement.

5.5.1 - Blank areas while text loads

After migration from static JSON to the database, delays in loading content caused blank spaces on the site resulting in a poor user experience. To fix this, I used i18next's fallback option with the `t()` function and created a static file with English text on the front-end. This allows the page to load instantly with English, and any translations from the database will then replace it when the fetch completes. This solution has the drawback of syncing of the two data sets, but this can be improved by automating the process. This solution works because front-end variables load faster than data fetched from a database as they are stored in the client's browser memory and are immediately accessible.

5.5.2 - Changing the sites format caused everything to reverse

I initially set the `documentElement.style.direction` CSS property to the direction provided by `i18next` to handle the change of format when switching to a RTL or LTR language. However, as I had created two versions of the `languageBar` component, one for each direction, this solution only flipped the component direction, for example when Arabic was selected, the page would read RTL, but the language bar, set to RTL, would then flip to LTR. To fix this, I passed the `dir` prop to the container holding the content and not the language bar.

5.5.3 - Conflict when updating database

After implementing the back-end and database for the common texts, I received an error (see Figure 31). This was caused by selecting a language like French, then English and then French again. The issue was that the database schema had no requirement for a unique `language_code` and `id` combination, resulting in multiple rows with the same `common_id` and `language_code` that caused a conflict.

I resolved this by adding a `CREATE UNIQUE INDEX` command in the schema for the `common_i18n` and `content_i18n` tables, to ensure the indexed columns (`common_id` and `language_code`) contained no duplicate values (see Figure 32).

5.5.4 - Phone numbers reversed

During bug testing, it was found that when a RTL language was selected, direction-specific elements such as phone numbers were being reversed, making them incorrect. To fix this, I used the global attribute `dir` from the HTML Standard. For all phone numbers

and other direction-specific content, I set the `dir` attribute to `ltr`, causing them to remain LTR when translated to a RTL language.

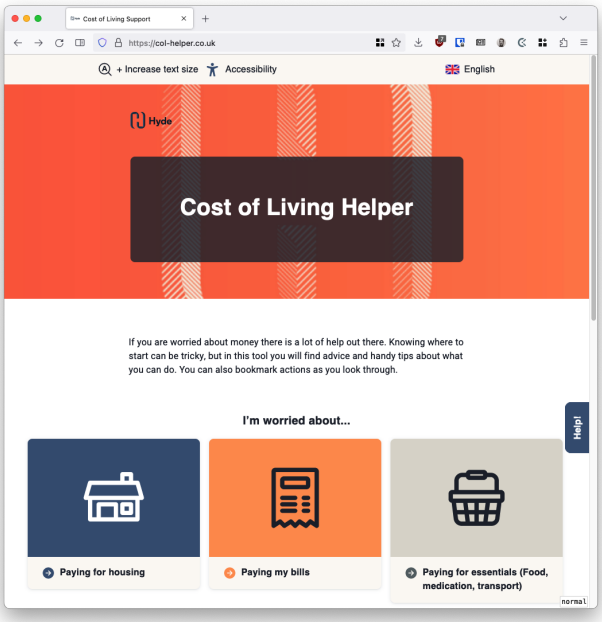
5.5.5 - Extended language codes

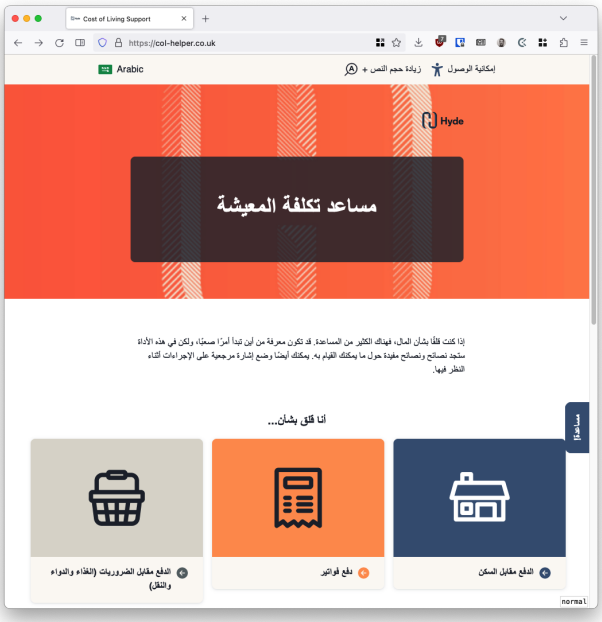
I discovered a bug during testing that stopped language translation from working for browsers with multi-part language codes. For example, I found that French `fr` worked, while Swiss-French `fr-CH` were not being detected correctly. The language detection only detected single language codes like `fr`. To fix this, I sliced the first two characters of the language code to ensure it was correctly interpreted by the translation algorithm (see Figure 33).

6 - Conclusion

6.1 - Summary

My contribution to this project, the language translation feature, was technically successful. Users were able to select from 67 languages for instant translation, presented in their native direction. The solution I researched and implemented allowed for efficient translation of both existing and future content while saving cost. The implementation followed company procedures and resulted in easily maintainable code. The tests I wrote ensured future updates did not break the feature and would alert developers to any issues prior to deployment.





`↑ Deployed CoL-Helper`

6.1 - Recommendations

I recommend the site's language translation functionality be improved by adding the following features: caching translations JSON in local-storage for faster page loads and adding a loading bar to show the user something is happening when they select a new language. These features will enhance the user experience by making the page load faster and providing a clear indication that the translation process is taking place.