Requisit Project Document

Group 1: Anna Sapsford-Francis, Eleanor Crossey Malone, Margalida Bover & Mariam Hussein.

Introduction

Background

The core piece of software used in public libraries is a Library Management System (LMS). This handles processes such as borrowing and returning books, browsing the library catalogue, placing borrower holds on books to be transited from other branches, and general inventory management. It handles cash transactions for printing, fines and lost books. It also includes some CRM-like features such as registering new borrowers and updating personal information, providing borrower checkout histories and the ability to attach staff memos to customer profiles.
Libraries NI, the largest single library authority in the UK, currently uses the native Sirsidynix Symphony Workflows LMS. Examples of competitor softwares include Sierra by III and the cloud-based Axiell Spark. Libraries often make use of web applications to add functionality that is not included in the Library Management System (LMS). The Papercut web app is used for print server management and MyPC offers booking and moderation facilities for public access computers.

The problem

If the library service does not have a copy of a particular work in its catalogue, a borrower can request that the library purchase the work. At present, this is a mostly analogue process, since neither the LMS nor any of the web applications used by the library service offer this functionality. The process currently in place is:

1. A borrower provides a member of staff with the title and author of the work in a library branch
2. The member of staff searches internet resources such as Google Books, Amazon, FantasticFiction and WorldCat to gather bibliographic information on the most appropriate edition
3. The member of staff manually enters this information onto a Request to Purchase form along with the borrower's name and membership number - this can either be done in a word processor or in handwriting
4. The member of staff scans or saves the completed form, then sends it as an email attachment to the stock team. The filename and email subject line must have a particular format, so that requests can be sorted alphabetically by the stock team
5. A record of this request is manually added to an Excel spreadsheet by the stock team. The spreadsheet is viewable on the staff intranet and has a status column which must be checked if the borrower later inquires as to the status of their order.
6. The status is updated by the stock team in line with their decision. The request may be approved or declined. There is no notification, so the status must be regularly checked by branch staff.
7. If the book is ordered, the stock team will add it to the catalogue, place the borrower's hold on it and send it to the branch for collection.

The present process for requesting new stock occupies a disproportionate amount of staff members' time, often over a long period. The process of submitting the request is outdated and involves the manual copying of information, leaving a lot of room for human error.

The slow speed at which purchase requests are processed adds to the overall wait for library users. Because requests are recorded on an Excel spreadsheet rather than in a database, there is little scope to expand the functionality to include status notifications to staff or borrowers. This means status updates can only be obtained when a borrower emails, phones or visits a branch to inquire as to the status of their request.

A web application that takes advantage of an external API, offering a single and fairly authoritative source for bibliographic information, would significantly streamline the process of placing borrower purchase requests by allowing searching and auto-filling bibliographic data onto an order. No single source can account for all possible requests, for instance some self-published books lack ISBNs and are generally harder to find information on, so there will still be occasions when the order data will still have to be provided by branch staff. However, even in these cases, the existence of an online form which submits the information directly to a database accessible by the stock team will represent a marked improvement in the efficiency of the process, freeing staff to accomplish other tasks.

For library users, the improved speed of the process will mean that staff are likely to be able to place the order while the library user is still in the branch, rather than having to take the information and assure the library user that the order will be processed later. It will also reduce the overall wait time, and offers the longer-term possibility of linking to an existing member database and setting up automatic email or SMS notifications.

The expected outcome is an intuitive, modern and public-sector appropriate web application that is a valuable addition to the existing toolbox of technologies used to deliver the public library service.

Specifications and Design

Technical Requirements

The application requires a front-end interface where librarians can log in and search for bibliographic information by key terms utilising Google Books API. Search results should be retrieved from the API and librarians can select any of those results to view the book in greater detail. They should then be able to use it to auto-populate an order form. It must also be possible to manually populate the form in case the information cannot be found automatically. When the form is submitted this information should be saved to a database. Members of the order team should then be able to log in, view the orders and update their statuses.

React Hooks are JavaScript functions that allow the developer to use state and lifecycle methods within functional React components. Consequently, React applications are made without using JavaScript classes.

```
import React, { useState } from "react";
function ContactUsForm() {
 const [isEmailValid, setIsEmailValid] = useState(true);
 const [isTextValid, setIsTextValid] = useState(true);
 }
```

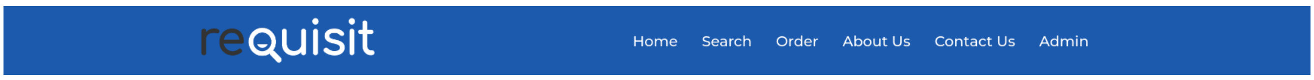
In this case, the component ContactUsForm has two pieces of state isEmailValid and isTextValid which carry the information of whether certain parts of a form are valid or not. This is used to show the user where the input errors are, so they can fix it.

Non-Technical Requirements

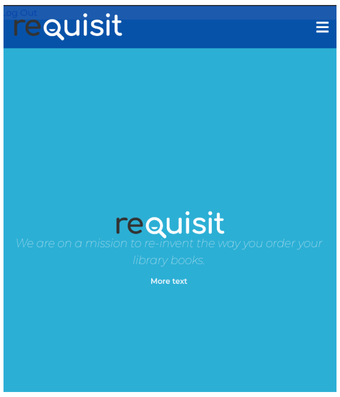
The application should conform to the expectations of a public sector application. It will be used by a wide range of users with different technical backgrounds, so it should be intuitive and easy to use. The application should be usable on a desktop or laptop. Given a longer timeframe, the application would also be optimised for tablet use.

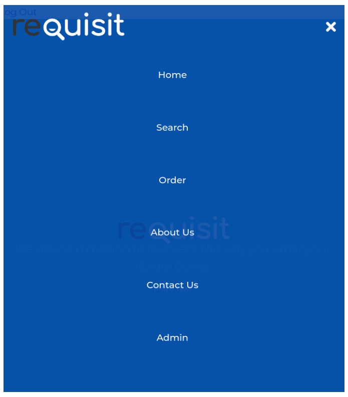
Design

The *law of common region*, one of several from the Gestalt psychology school, simply asserts that if components on a page are clustered together closely, they are regarded as being connected to one another.
With borders, backgrounds, or spacing, you can achieve this. For instance, a navigation bar or menu is typically created by grouping various navigation links:



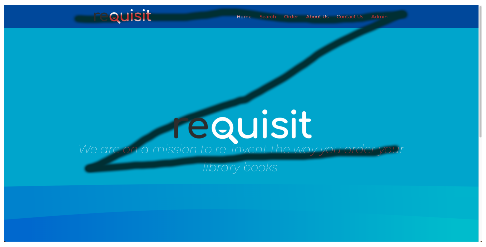
We had to use this approach carefully since it is all about composition and spacing.
In UI development, *Jakob's law*, which was developed by the co-founder of the Nielsen Norman Group, Jakob Nielsen, encourages the use of logical and familiar scenarios. Users typically like and expect a website to operate in the same way as others they are already accustomed to.
Each of us develops mental representations of the conventions that surround websites. This allows consumers to concentrate on their goals rather than figuring out how to use a strange UI.
This indicates that we should avoid overwhelming them with novel situations and instead stick to what they are acquainted with. For instance, clicking on a "burger" icon typically opens a menu:

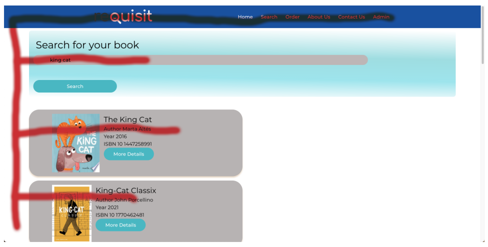

requisit
requisit
We are on a mission to re-invent the way you order your library books
More text



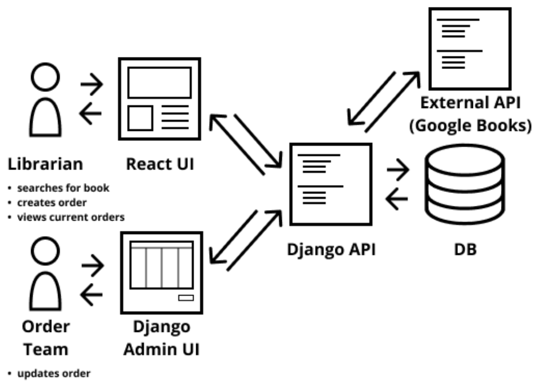
The choosing of colours on the opposing sides of the colour wheel, which are typically complementary colours, is known as a derivative colour combination. In order to increase the complimentary and contrasting colour scheme, a cool colour is frequently mixed with a warm colour. Contrarily, it is advised in the theory of colour that while creating interconnected colour schemes for a branding, it is crucial to select one colour that will serve as the design's focal point. In this project we decided for blue as it is known as a colour that transmits trust.

Although the 'F' pattern is frequently observed, it is not the only one. Yet another key design principle is the "Z" layout. This is when the eye scans in a hypothetical horizontal line from left to top right. A fictitious diagonal line is then drawn from the top of the page to the bottom of the next page. Finally, it trails across to the right once more, creating another horizontal line.
Whereas the "F" layout typically performs best for landing pages, the Z-pattern is typically preferable for sites with very little information where the call-to-action is the major takeaway.





Architecture



Librarians will access the application through a React frontend hosted within a Django template. The librarian will be able to log in via a static Django page that will redirect them to the React frontend once they are authenticated.

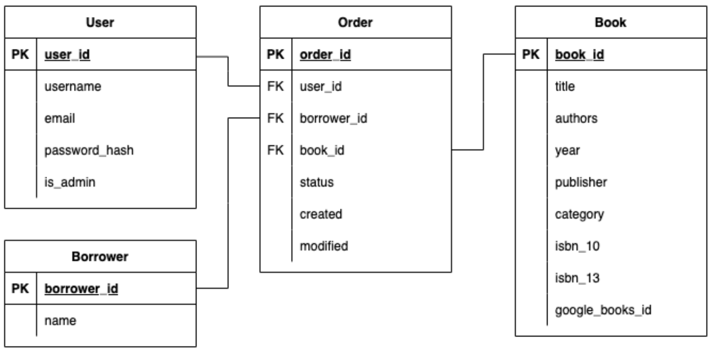
When they search for a book the React UI will make a request to the backend Django API, which will send a request to Google Books API to get book search results. The Django API will filter out unnecessary fields, flatten the JSON results and return them to the React frontend.

When the librarian selects a specific book, React will again send a request to the Django API, which will request book details from Google Books. Then the Django API will process the JSON data and return it to the React frontend.

When the librarian creates an order, a post request containing the book and order details will be sent to the Django API. The API will then validate the data and, if valid, save the data to the relevant tables of a SQLite database.

The order team will have access to the application through the Django admin view, which is a configurable admin system that Django provides. The team will be able to log in, view the current orders within the database and then update them.

Database



The SQLite database is created using Django's built-in tools for database models. Serializers determine which data will be accepted at the Order endpoint and where in the database it should be stored.

When an order is submitted, the Order table records the date and time created and last modified using Django's timestamp feature. The status column is automatically assigned the value 'Pending'. If the status or any other information is subsequently changed by staff in the Django Admin view, the 'modified' timestamp will update.

The Order serializer also takes book and borrower information from the JSON data and stores it in the corresponding tables. The Book table has an auto-incrementing Primary Key while the Borrower table accepts the Borrower's membership number as a Primary Key. These are linked to Foreign Keys within the Order table. The database allows for the addition of a User table containing the details of staff members with and without admin privileges, although for demonstration purposes, there is only one registered admin user.

Implementation and Execution

Team Roles and Responsibilities:

The software development approach for Requisit was a mix of Agile and Waterfall achieved by splitting up the project into the following strengths and skill-based roles:
Anna - *Tech Lead:* They are responsible for choosing the type of stack used for the project and connecting the back to the front-end.
Eleanor - *Product Owner*: They are responsible for providing insights on current problems faced with LMS and the project requirements to address it.
Margalida - *Design Lead*: They are responsible for the aesthetic choices and design of the interface. This includes the selection of a colour palette, layout, and interactivity of the website.
Mariam - *Project Manager*: They are responsible for the management of the project. This includes organising project agendas, roles, expectations, and the tracking of progress for Requisit.
For the planning for Requisit, each member provided an insight into their language proficiencies, timetables, work commitments, etc. to ensure the implementation process aligned with their schedules.

Development Approaches

As part of the development of Requisit, the approach used has been a mixture of Agile and Waterfall methodologies.

Waterfall Development was used to plan the project collectively, including Requisit's requirements, design, analysis, and timeline. This ensured realistic timelines and expectations on the 4-weeks' workload to create Requisit. The use of a chronological approach meant the team could plan and mitigate issues that may occur. The foresight from Waterfall's planning approach ensured that decisions and changes made were still in-line with the client's functional needs. However, with a smaller team, Agile methodologies were implemented to ensure quick turn-arounds, flexibility with changing the final product, and consistent product development.

As part of the planning, monitoring, and development of Requisit, the team conducted regular meetings. At the start, these were opportunities to plan and develop the Minimum Viable Product (MVP) of Requisit, group responsibilities, and a timeline for this project. Throughout the implementation of Requisit, daily stand-ups were scheduled to gain an overview on current tasks, issues and outcomes. This was an opportunity to raise questions and to check the timeline of Requisit's tasks. The stand-ups provided an opportunity to change any implementation decisions whilst staying on track. Alongside this, the Product Owner and Project Manager would analyse the Requisit currently and provide feedback to all members, including themselves, on potential key changes. Requisit's features were developed using an MVP to ensure that within 4-weeks, a functioning product that met Libraries NI's current issues with realistic sprints was produced. Requisit's MVP is a functional app that can *search and display book results, which then auto-fills a form to be stored into a database*. Using an MVP and the Product Owner's insights meant that decisions made about the project early on stayed true to solving the client's problems, whilst allowing for flexibility in adapting any problems.
Staying true to Agile Development, team Requisit took on a 'Sprint'-like ticketing approach, where small tasks to build features over 1-3 days after evaluating current progress and comparisons to initial planning were conducted. These Development methods allowed for a smoother Implementation process as the project had clear expectations of the product, whilst being considerate of everyone's commitments.

Implementation Approaches

The main implementation process has been incremental development, which ensures every version of updated code is a usable functional product. The Tech Lead and Design Lead have maintained the 'main' branch of Requisit's GitHub repository by testing the user interface and back-end of Requisit with every pull request.The usage of vertical increments has meant each ticket issued via GitHub's 'Issues' tool allowed new features to be built whilst the back and front-end were connected from the start of the project.

Further Implementation practices included pair coding, which allowed those struggling to implement the next stage of code to be supported by a more experienced coder, such as the Tech Lead. This consisted of pairs calling to write cleaner code and functional lines that meet the requirement of the sprint. It allows for the team to talk and troubleshoot issues as simply as possible. This also allowed everyone to understand the front and back-end code without confusion. Alongside this, day-long sprints were used to set realistic targets and workload for team Requisit to break the project into manageable blocks of work to reach smaller goals. As this was a 4-week intensive project, it made sense to allow team members to focus on their individual strengths and developing strengths through day-long tasks. This process, when coupled with the incremental approach ensured any implementation decisions and challenges were made based on the latest and most functional code repository.

Implementation was supported with the initial project planning. Having a clear awareness of what has to be produced based on the MVP. Several decisions on features were made based on the remaining timeline and the workload of the team. An example is the stretch goal of having separate users for the librarians and stockists, so that stockists can log into the database of Requisit, whilst librarians can log into the main order interface. The decision to shelve this feature came after comparing what the back and front-end teams had to modify and troubleshoot for this feature. Based on the extensive work that had to be done, team Requisit made a group decision to prioritise more important features to ensure we reached our MVP on time. This approach of comparing current timelines, our goal to an MVP, and the time dedicated to a feature allowed team Requisit to realistically work towards key goals with ease.

To conclude, the usage of an MVP and daily stands-ups allowed flexibility across the team to realistically measure achievements based on each iterated feature added to the main Requisit repository, whilst working as a team to measure how long additional features would take. Having both a functional MVP as a goal, made planning and deciding on stretch goals, such as a confirmation page, easier to coordinate.

Tools & Libraries Used

Management and technical tools were used to work towards the MVP. The Project Manager used Trello, the management features on GitHub, and the Figma Wireframe to ensure members stay on track.

With Trello and GitHub, code reviews were assigned during a pull request on a dedicated Trello Card, which was then highlighted via GitHub tickets. This prioritised major bugs during the implementation of Requisit, and to assign less urgent code reviews later on. Trello's assignment feature ensured that tasks assigned after a stand-up were notified to members' inboxes. This achieved transparency on everyone's key tasks and responsibilities for the day, and affirmed each member on their 'mini-sprint'. GitHub's features were useful in comparing code and tracking the progress of each member, as GitHub allowed the latest changes to be reflected in each branch, thus strengthening flexibility to pair-program.

The usage of Libraries has been limited for two reasons. First, the Requisit Team wanted to challenge themselves on creating a functional front-end by using pure React. Many members are new to full-stack development, so using React for the front-end features reinforced lessons learned during CFG. The other being timeline issues with implementing Libraries such as Bootstrap, MaterialUI, etc. Implementing libraries means any existing CSS stylings would have to be adjusted for each component and page. After key considerations of timelines, front-end libraries were scrapped for this project. This was made with comparisons to the wireframe to ensure the final product followed the wireframe as closely as possible.

Implementation Challenges

During implementation, key challenges came to fruition. Communication regarding key tasks and achievements became unclear after stand-ups. At the start of Requisit, the sheer amount of components and functions that needed to be built meant some members did the same tasks by accident. To ensure consistency in communication, team members were required to repeat what key tasks they had at the end of each stand-up. This was then logged into a GoogleDrive folder of meeting agendas, then placed onto a Trello Board for everyone to track. Alongside this, the management of work and key changes was difficult at the start due to the various workflows and team splits occurring (back vs front-end). Thus, the team was sometimes out of touch with current code and what main features were being implemented. To help with this, an extended weekly meeting provided team members to have a file by file walkthrough of existing code and its purpose. This ensured both the back and front-end team knew what was being coded and what each file did on each side of team Requisit's architecture.

Finally, pair-coding provided an exciting opportunity to work together on code. However, it became confusing once merging changes made to a branch using Git as any changes were sometimes overwritten or accidentally deleted. For this, the members took turns learning how to use Git, or finding easier solutions such as GitHub Desktop and VSC Extensions, to ensure pair-coding maintained any changes made by both coders.

To summarise, the implementation of development of Requisit happened with the collaboration of all 4 team members with the support of extensive project management tools, Agile and Waterfall methodologies, and regular communication and support.

Testing and Evaluation

Testing Strategy

Due to the short timeframe, the team relied on manual testing during the development phase. Before a new feature branch was merged all aspects of the new feature were tested manually by the developer and the code reviewer. Once the branch was merged, the code reviewer carried out manual regression tests as well as manual tests of new features. For branches in development for a longer period of time, developers merged main into the branch before testing.

Functional and User Testing

Towards the end of development, unit tests were created to test the backend Google Books API logic. Tests were also created for two of the database models (Book and Borrower). Ideally, additional tests would be created for the remaining database models (Order and User) as well as tests for post requests to the database endpoint /api/orders/.

It would also be advisable to create frontend tests, possibly using Jest and React Testing Library, to test the frontend UI.

System Limitations

Requisit uses Django’s default SQLite database. While SQLite handles limited data sufficiently and has allowed us to achieve our MVP, a client/server database such as MySQL could be more appropriate if the app were to scale up. SQLite locks the database for writing, which means that the performance could lag in a scenario where many requests are being processed at one time. Were the app going into production, the order form functionality could be expanded to pull data from an existing library member database, which could also be used to obtain member contact details and send email and SMS updates.

The purpose of the admin site is to allow the stock team to view placed orders, update the request status in line with their purchasing decision, and edit order information. While the admin site performs these functions, it would be positive in hypothetical future iterations to style the admin site in a way that is more harmonious with the front end. It would also be necessary to have a separation of user permissions and create two user types, staff and admin, to represent different roles, and include User IDs in Order records. Redux could be added to the frontend to simplify state management and make the code more readable. The addition of Redux would also allow easier implementation of props across several components and pages. The ***‘book’*** prop, which contains all bibliographic information, had to be fed through 4 components to reach the Order Form on Requisit. With Redux, the management of states and props allows for smoother integrations of front-end code.

To balance using our existing abilities with learning new skills, we decided in the planning stage to have the Django backend call the Google Books API. Given the short duration of the project, this decision was correct as it allowed us to achieve our MVP. However, can be noted as an area for improvement, since calling the API directly from the frontend would give a performance enhancement and could be done in fewer steps.

Since the Google Books APIs do not provide all the required information with the search URI, it was necessary to make a second call to the volume URI to complete the order form. It may be beneficial to consider using other resources such as the Open Library APIs. These would allow for more sophisticated search features, such as the ability to search by work and then use the Works API to display a list of all editions of a particular work. It also includes a wider range of physical media, including audiobooks on CD ROM and Playaway. Along with new UI features such as filters on search results, which could include filtering by language, genre or decade published, these tools would help library staff to more easily find less well-known titles.

Conclusion

Our aim was to improve upon the manual book ordering process used by Libraries NI. We were conscious of the short time frame, so our intention was to make a minimum viable product as opposed to a fully-featured application. In that regard, we believe we were successful. We created an application that allows users to log in, search for a book, auto-populate the details of that book to an order form and then create an order that is saved to a database. These orders can then be tracked and updated in Django's admin view.

The process of developing the application was a period of intense learning for the entire team. We learned how to collaborate on GitHub, including creating pull requests, performing code reviews and merging branches. The frontend team learned React, Webpack and Babel. Whilst the backend team got to grips with Django and Django Rest Framework. We also learned how to work together as a team. In our regular meetings we discussed which features we would implement next, what our deadlines should be and we negotiated workloads and watched out for scope creep. This project with its relatively short time frame, has given us the confidence and enthusiasm to tackle longer and more complex projects.