



Dungeon Desk – Developing a co-operative real-time Web Application with React and Node.js.

**An accessible tool for playing Tabletop Role-Playing
Games with others.**

Taylor Jordan

CandNo: 262835





Contents

1. Introduction	4
1.1. Table-Top Role-Playing Games	4
1.2. Objectives and Existing Solutions	5
1.3. Relevance and Motivations	6
1.4. Background Research	7
2. User requirements	8
2.2. Requirements Analysis	8
2.2.1. Creating a table	8
2.2.2. Inventory functionality	8
2.2.3. Extension Requirements	9
2.3. Requirements Specification	9
2.3.1. Web Server and Back-end	9
2.3.2. Database	9
2.3.3. Front-end	9
3. Project Plan and Processes	10
3.1. Management Methodology	10
3.2. The Original Plan	11
4. The Software Stack	14
4.1. React.JS Overview	14
4.1.2. Declarative VS Imperative	17
4.1.3. Functional Programming in React	17
4.2. Node.JS Overview	18
4.3. MongoDB Overview	18
5. System Design and Considerations	20
5.1. The client-side	21
5.1.1. Visual Design	21
5.1.2. Component Hierarchy	23
5.1.3. State as Data	24
5.2. The Server-side	27
5.2.1. Real-time synchronization	27
5.2.2. Server State Management	27
6. Implementation	29
6.1. Inventory Features	29





6.2. Visual Design.....	30
6.3. Front-end state management.....	32
6.4. Feature Requirements	34
6.4.1. The Item Set.....	34
6.4.2. Inventory Pages.....	34
6.4.3. Creation of tables.....	34
6.4.4. Real-time synchronization	36
6.4.5. Undoing Changes.....	38
6.4.6. Sub-Inventories.....	38
6.5. Mobile Adaptations and Considerations	40
6.6. Web server set-up	41
7. Conclusion and Future Work	42
7.1. User feedback results	42
7.2. Future Work.....	42
7.2.1. Back-end	43
7.2.2. Front-end.....	44
<i>Appendix</i>	45
A1. Professional Considerations	45
A1.1 BCS Code of Conduct.....	45
A1.2. Ethics Considerations.....	45
A1.2.1. User Testing.....	46
A1.2.3. Rights of third-parties.....	48
A1.2.4. Signed User Testing and Compliance Form.	49
A2. References	53





1. Introduction

Computer software is ubiquitous today -- computer programs influence our work lives and personal lives, often in the form of apps and services. Software has revolutionized how we write, organize and communicate – for example, word processors, digital spreadsheets and messaging apps. An especially potent use for software is transforming tasks that are often mundane and tedious to make them quick, easy and even engaging. In this project, I will show how I have applied this to the popular hobby of Tabletop Role-playing Games, by building a complete, full-stack web application, to improve an aspect of the hobby that I believe warrants a robust digital tool.

1.1. Table-Top Role-Playing Games

A Tabletop Roleplaying Game (TTRPG) is a cooperative game, where the players adopt the role of characters in a story, and interact with the game world through a Game Master (GM) who governs what happens in the world. TTRPGs are extremely popular, and the hobby is enjoyed by millions worldwideⁱ, providing an opportunity for people to socialise, build connections, express themselves through their characters and exercise their creativity to build a cooperative story, as well as partake in some friendly competition. Those who frequently engage in the hobby can range from combat and mechanics fanatics, who optimise their play to maximise their damage output in encounters – to those who play solely to craft a dramatic and heart-wrenching narrative, making decisions based on what would lead to compelling story moments.

This hobby is an important part of many people's lives, and studies suggest that partaking in a TTRPG has the potential to improve someone's mental wellbeing, by providing opportunities to connect with others and express themselves freely that the player may not otherwise haveⁱⁱ.

Each individual TTRPG proposes a system for how the world operates and how players can interact with it, with a large emphasis on character building and encounters. For example, the concept of a “class”, in Dungeons and Dragons, determines what a character can do in combat, the character's strengths and weaknesses and often serves to guide what role the character will take on in the storyⁱⁱⁱ.

Some TTRPGs are very intricate and complex, with players and Game Masters having to familiarise themselves with hundreds of rules and guidelines to fully experience the game. The most popular system currently; Dungeons and Dragons (5th Edition), is a prime example of this. The volume of rules, abilities and content can force a large mental load upon the Game Master and the players^{iv}, and this may increase exponentially the more players there are at a table. A pen-and-paper approach to managing games can be difficult and exhausting, and some believe it creates a barrier for entry that can turn some away from the hobby^v.





Digital tools have been developed both by fans and creators alike to help manage this complexity. Some of these tools are open-source and free to use, while others are paid and licensed products that belong to the publisher.

These tools can be immensely helpful for managing a table – they reduce the mental load that is placed on the GM and can help inexperienced players experience the game in a more fun and intuitive way. However, despite the many digital tools that exist for these games, several of them don't quite offer what is needed at a specific table.

In the next section I will discuss the shortfalls of these tools, and how I think that the development of a new, general-purpose tool could be a great asset to have at table, and my motivations for this project.

1.2. Objectives and Existing Solutions

Most games (video games, sports games) use a rigid set of rules, in which violating them ruins the integrity and enjoyment of the game, especially if the main incentive is to win.

TTRPGs tend to differ. While some systems guide players to a more competitive and rigid style of play, the hobby is often deemed a storytelling medium first, with a game on top of it, and this opens ample room for rule-bending and customisation^{vi}. Some players even develop their own systems that they use in combat or change aspects of an existing system that they are using. This freedom is what draws a lot of people to the hobby, but many of the popular digital tools struggle to accommodate this demographic of players.

For example, the official companion software for Dungeons and Dragons: D&D Beyond, is very popular amongst D&D players and used very often to help manage games. It has an inventory system that allows players to manage their character's items, including options to add in-game or custom items^{vii}.

However, D&D beyond is designed solely for Dungeons and Dragons. Many features of this software require a subscription to access, including adding "Homebrew" (fan-made) content to games, and users who run games with an excess of home-made content have sometimes expressed dissatisfaction online with the process for making these customisations^{viii}.

Roll20 is a free companion software for several established TTRPGs, including Dungeons and Dragons and Call of Cthulhu. It primarily serves as a virtual table-top; a program designed to simulate the experience of playing on a table using software, primarily to enable users to play TTRPGs online with people from across the world.

However – as of right now, Roll20 has no robust solution for handling player inventories. Players are given the ability to write scripts, which can be great for customisation, but requires prior programming knowledge^{ix}.

The companion software available to a player often influences which TTRPG they decide to play. Inexperienced and experienced players alike rely on this companion software to access rules and content about the current state of the game, which can be immensely helpful to maintain the flow of combat and story moments^x.





For example: a player may decide to use a special item to influence the outcome of an interaction - but what if they can't find it, or can't remember if they still possess it? Maybe it still appears on a spreadsheet, but they have a vague memory of discarding it in a past session.

While these moments are often trivial and easily solved, some tables take a very immersive approach to roleplay and combat, where those kinds of events can make a tangible difference to the experience.

Many table-top RPGs do not provide companion software, and this can lead to players hesitating to try those systems - especially for groups that play over the internet. I hope that this project will provide universal tool to be used across every system.

1.3. Relevance and Motivations

This project will test many of the skills that I have developed over the course of my degree. The front-end will be designed to provide the optimal user experience, with an emphasis on being responsive, aesthetically pleasing, and self-explanatory. An understanding of multimedia design and human-computer interaction will be essential to provide a satisfactory user experience.

Developing a full-stack web application within a given timeframe will test my ability to manage my time and workflow effectively, and these skills will be essential to ensure that this application meets the requirements of the target users^{xi}. I must keep an accurate track of progress, adapt to any unexpected setbacks and employ a rigid methodology for making progress during development.

For me personally, building a complex full-stack web application will be an invaluable experience. This project incorporates almost every part of my degree and will hopefully be a great representation of what I have learnt. Through this project, I can also further my skills in the industry standard technologies that are used to build large scale web applications and greatly expand the knowledge that I currently possess^{xii}. This project will be a great addition to my portfolio as a developer and will demonstrate a desirable skill set to potential future employers.

I also hope that the outcome of this project is a tool that could serve as a positive contribution to the TTRPG space; to allow players and Game Masters to tailor their experience to what they want, to reduce the mental load on players and make TTRPGs more accessible to those who are discouraged by the complexities involved, and to allow players the freedom to put their own spin on the system they are using. Or depending on the user, simply a more comprehensive way to manage their inventory in their role-playing games.





1.4. Background Research

To form a clear set of user requirements, I conducted some research in the form of an anonymous survey – targeted at those with almost any level of investment in TTRPGs.

The survey collected data on how participants preferred to manage their resources while participating in a game. Many participants reported that they simply kept track of their resources (items, character abilities, notes, etc) using either pen-and-paper, or note-taking applications. Some players described that in their games, resource management was not very important and thus, did not desire a different solution. Others mentioned that they had often played in games that are very resource-focused and less free-form and in such cases, have been dissatisfied with the experience of manually taking notes to describe their resources.

A few of the participants had experience in the role of Game Master for Dungeons and Dragons – Many of those participants expressed demand for a digital tool that would allow them to see the items and resources of their players and to be able to control and modify them. This would allow them an easier time cooperating with the players while also providing new and creative ways to implement aspects of gameplay.

Two participants claimed they had almost no experience playing TTRPGs, instead enjoying the medium by watching others play TTRPGs on various video series (e.g. Critical Role^{xiii}) and playing video games that are heavily influenced by TTRPGs (for example, Baldur's Gate 3^{xiv}). In the extra notes, these participants expressed that they feared the complexity of TTRPGs and hesitated to participate due to the diligence required to follow the rules of the game.

The results of this survey outline a clear set of requirements for an application that could potentially improve the experience of TTRPGs for the participants.





2. User requirements

From my own subjective experiences, I believed a tool like this could be very helpful for making TTRPGs more fun and immersive, by simplifying some of the error-prone and tedious processes. The following user requirements are drawn from the survey feedback and seeking the opinions of users online.

2.2. Requirements Analysis

The following requirements describe in detail how the features of the program should interact with the user.

2.2.1. Creating a table

Upon accessing the homepage, users should be offered a simple and self-explanatory way to create a table.

1. The user should be able to create a table, where they are navigated to a different page that allows them to name their table
2. When their table is successfully created, they should be offered an ID that allows them to connect to their table, as well as a URL that can also be used to connect them to their table.
3. If the creation of the table is unsuccessful (e.g. DB is down), an error should be displayed to the user.

2.2.2. Inventory functionality

The application will provide an interactive means to allow users to manage their Inventory in games.

1. The user should be able to create an item by clicking a blank space in an inventory, or the item sidebar. This item may be given a name, weight, value, and icon.
2. The inventory should be displayed as a grid. There may only be a certain number of grid slots visible per inventory, so “Pages” will be implemented.
3. Any items added by users will appear in the “Item set” section on the to be added to anyone’s inventory.
4. Users can drag and drop item instances to other slots in their own inventory. They may also drop them into another player’s inventory.
5. Sub-inventories can be made - such as a pouch, bag, or any kind of container.
6. Users can attach “notes” to specific item instances in their inventory.
7. An item instance has properties, and these properties can be changed.





8. A search bar at the top of the inventory will allow users to input keywords, to filter through all pages of the inventory for a specific item.
9. Pages may be renamed and favourited. If a page is favourited, right clicking an item will prompt the option: “move to: [favourites page]”
10. An item instance can be removed by right-clicking and selecting delete.
11. Users can create, remove, and rename player inventories.
12. The inventory data must update in real time – clients on the same table must see the same information.
13. The user interface for the application must be suited for both mobile and desktop devices.

2.2.3. Extension Requirements

Some additional features for further in development:

1. In this system a player character will simply be an entity with health that can act in combat.
2. Players can add optional stats and notes to their character that fall under the “character profile” tab. This will allow the user to enter the name and value of the statistic.
 - a. A desirable goal would be to allow the user to enter expressions that reference other statistics that the character has like a digital spreadsheet.

2.3. Requirements Specification

Meeting the requirements set out in Section 2.1 will require a well-designed and stable system. This system will have various hardware and software dependencies. The below specification will describe how these requirements will be met and the resources these solutions will depend on.

2.3.1. Web Server and Back-end

Due to the nature of this application, a web host will be required. This web server will be responsible for performing all the back-end operations of the web application and housing the database of user tables.

2.3.2. Database

MongoDB will be used to manage the backend database. This database will house all the information relating to the tables and user accounts. [14]

2.3.3. Front-end

The front end will be developed using React. Good knowledge of CSS and JSX (JavaScript XML) will be required to achieve a pleasant, user-friendly design, as well as visual design principles and modern web design standards.





3. Project Plan and Processes

Building multi-part projects like requires a rigid time management methodology with a clear set of objectives. The following chapter describes the processes involved with ensuring the progress of development is stable and that the requirements will be met by the deadline.

3.1. Management Methodology

In software engineering, project management methodology primarily falls under two main categories: Agile and Waterfall.

In Waterfall project management, the entire process is broken down into linear, sequential phases. Each phase of the project depends entirely on the completion of the previous phase of the project – a phase cannot begin before the previous phase concludes, and it is not possible to go back to a prior phase. If amendments must be made to a previous phase, the entire project must be restarted^{xv}.

The Waterfall methodology is rooted in construction and manufacturing and provides a rigid framework for progress. It is best suited for projects that have a set of stable and unchanging requirements, and where there must be a single finished product without the possibility for revision. Some software projects require this stable guidance – especially with large workforces developing systems where failures and mishaps can have extremely costly or even deadly implications^{xvi}.

Many software projects nowadays require a much more flexible approach, however. The linearity of Waterfall does not accommodate for changes in user requirements and simultaneous workflow. Instead, software projects are often iterated upon, even after the product is in the hands of its target users. This requires a much more flexible approach to project management, which Agile methodology aims to fulfil^{xvii}.

There are two main parts to Agile Methodology: Scrum and Kanban.

In Scrum management, a project is broken down into two-week phases called sprints. Each sprint sets out the end goals for the phase and the responsibilities for each team member, with a ‘Sprint Review’ at the end of the sprint for gathering feedback and discussing improvements that must be made.

Kanban allows for effectively managing the workflow of a project using a visual pull-based system (a Kanban board), where tasks are categorized into their current stages of completion – Backlog, Ready (to be started), In Progress and Done. This helps to visualize any bottlenecks in the workflow and measure the overall progress of development^{xviii}.





For this project, I met with my supervisor between each sprint for a ‘Sprint Review’, using a Kanban board to manage the tasks for each sprint.

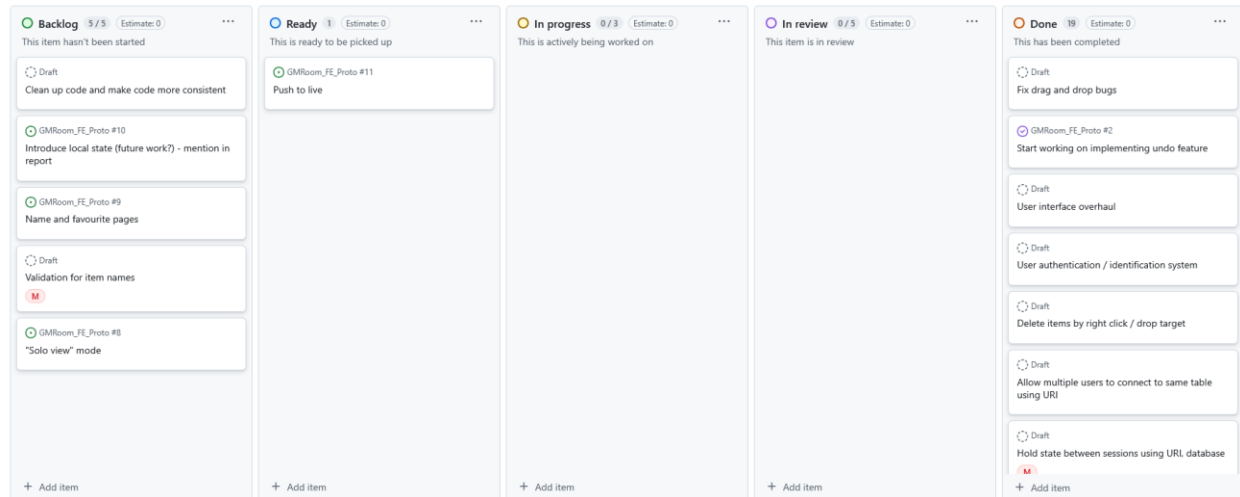


Figure 1: A Kanban board used to manage project tasks, hosted on GitHub.

3.2. The Original Plan

At the start of the project, I defined a list of tasks that were to be completed throughout the project. A project plan can be found for these tasks in the following section -- however, due to changes in the scope of the project later in development, many tasks in this list – specifically in regards to the “combat tracker”, are redundant:

- 1) Create high-level designs of the backend systems, including the database.
- 2) Develop front-end inventory system in react
 - a) Create a prototype design in Figma
 - b) Work on the essential react components to be re-used in the end implementation.
- 3) Develop database and back-end structures for player characters and users
- 4) Implement the back-end component for the inventory system in Java Spring.
 - a) Implement REST API for the inventory and character systems
 - b) Implement the front-end with the back end.
- 5) Develop front-end combat tracker in react
 - a) Create design prototype in
 - b) Work on the reusable react components
- 6) Implement back end for combat tracker
 - a) Develop data structures and REST API
 - b) Implement with react-front end.
- 7) Implement the multiplayer table system and user accounts into the backend database.
- 8) Implement the front-end for the home page and dashboard.
- 9) Implement the back-end for the home page and dashboard.





The next page displays a PERT chart to represent the timeframes and dependencies of each of these tasks.

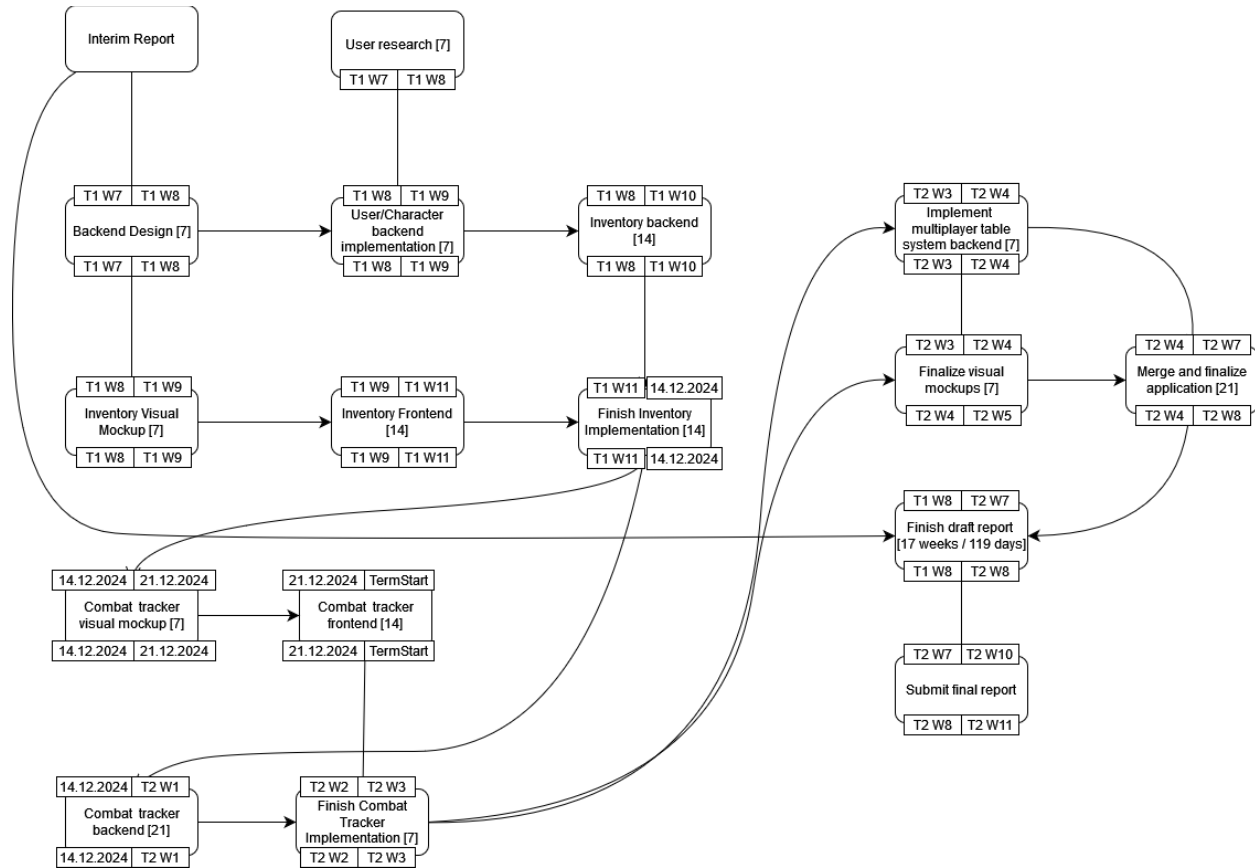


Figure 2: The original PERT chart from the interim report

There are many changes to the amend to this project plan:

- 1) "Combat tracker" feature is absent due to time constraints
- 2) Back-end server uses Node.js instead of Java Spring
- 3) "Visual Mock-ups" would not be made prior to implementation

While some of the intended features are absent from the finished application, the application does serve its original intended purpose. As development began, the priorities of the project changed and development became less linear. I decided to prioritize the back-end, working on the co-operative features of the application instead of adding more features. This allowed me to further my understanding of back-end development, and promote the cooperative nature of the project.

Below is an updated table documenting the sprints and objectives that took place through development.





1	<ul style="list-style-type: none">- Built inventory, item slot, item components- Implemented Drag-and-Drop functionality- Custom context menu	2 weeks
2	<ul style="list-style-type: none">- Item creator built (model box)- ItemSet component built	2 weeks
3	<ul style="list-style-type: none">- Item Properties can be set and changed- Quick Create option when right-clicking empty slots	2 weeks
4	<ul style="list-style-type: none">- Multiple inventories can be created and deleted- Item Counter- Items can be stored on different pages- Available from remote web server	2 weeks
5	<ul style="list-style-type: none">- Container item type, with <PlayerInventory/> wrapper component- WebSocket endpoint for real-time synchronization- User Interface re-design- Search bar on Inventory- CSS media queries to ensure mobile-friendliness	2 weeks
6	<ul style="list-style-type: none">- Users can make custom tables, and share a URI to allow other users to connect (react-router implementation)- State consistency with MongoDB- Undo feature in progress- Reimplemented drag-and-drop for mobile device support	2 weeks

Figure 3: A table of the tasks completed at each sprint.





4. The Software Stack

Building a full-stack web-application will require the use of existing software frameworks for handling specific tasks. With the individual parts of the application being very separated and de-coupled, each part of the application will require different solutions. In this case, the chosen front-end solution is the React.JS framework, the back-end solution is Node.JS with the ‘WS’ and ‘Express’ libraries, and the database solution is MongoDB. This chapter will provide an overview of why I decided to choose these solutions and what they provide for this project.

4.1. React.JS Overview

“React.js” is a JavaScript framework for building user interfaces for web applications. For enabling interactivity with web pages, Web Browsers allow web pages to include program logic written in JavaScript, a general-purpose programming language, originally designed for interacting with the DOM (Document Object Model) in a document^{xix}. For example, a web developer could use JavaScript to update a HTML element when a user clicks a button, to show how many times the user clicked that button.

As web applications began to grow in complexity and scale, it became clear that JavaScript was slowing down the development process of these applications. The de-coupling of JavaScript logic from the HTML and CSS code that describe the content to be shown can introduce lots of repetition and redundancy that makes for a codebase that is difficult to work with^{xx}.

React is one of many libraries that reduce this complexity to provide a more intuitive methodology for building front-end applications. Instead of describing content separately from the program logic, React allows developers to write Components^{xxi}. Components inherently include logic and are primarily implemented as JavaScript functions that return markdown in the JSX language. The JSX (JavaScript HTML) language has almost identical syntax to HTML but allows for JavaScript expressions to be injected directly into the content. This speeds up the process of implementing dynamic page elements^{xxii}.

To demonstrate this, the next page features a slightly modified snippet of the code for the Inventory component, with annotations.





```
function Inventory(props) {  
  ...  
  let inventoryContents = appContext.inventoryContents  
  const renderInvSlots = (contents) => {  
    return [...Array(30).keys()].map((num) => (  
      <ItemSlot  
        ...  
        item={inventoryContents[num] ? inventoryContents[num] : null}  
      />  
    ))  
  }  
  ...  
  return(  
    <>  
      ...  
      <div className="Inventory">  
        {renderInvSlots()}  
      </div>  
    </>  
  )  
}
```

A state variable, passed down through Context.

The ItemSlot component takes 'item' as a prop and has it's own logic for handling input.

The output of renderInvSlots() returned alongside the page content.

Figure 4: A code snippet from the Inventory component

React components permit calling expressions directly in the markup, which allows us to update the page content when a state variable changes. Here, the visible content of the page will automatically update with `inventoryContents` as it is a state variable that is passed down through context.

To compare, here is how this may be implemented using vanilla JavaScript:





```
const itemGrid = document.getElementById('itemGrid')
```

```
document.body.onload = renderItems();
```

Calling the DOM to access an element in the HTML

```
function createItem(obj) {
```

```
    let slotImg = document.createElement("img");
```

```
    slotImg.src = obj.itemImgPath;
```

```
    return slotImg
```

```
}
```

```
function renderItems() {
```

```
    let slots = document.createElement('div');
```

```
    slots.className = 'itemGrid';
```

```
    [...Array(30).keys()].map((i) => {
```

```
        let newSlot = document.createElement('div')
```

```
        newSlot.className = 'itemSlot';
```

```
        if (inventoryContents[i]) {
```

```
            newSlot.appendChild(createItem(inventoryContents[i]));
```

```
        }
```

```
        slots.appendChild(newSlot);
```

```
    })
```

```
    itemGrid.children = null
```

```
    itemGrid.appendChild(slots)
```

```
}
```

After a DOM element is made, it has to be assigned to the parent component using `appendChild`.

The `renderItems()` function is must be called whenever there is a change to the inventory, otherwise the screen will stay the same.

Figure 5: A JavaScript implementation of the React component





JavaScript must use the DOM API to reference the container that houses the items on the page, instruct the program to create the element required, then add it to the page. This is verbose, inflexible and time consuming to write, with many instructions that seem unnecessary and redundant. None of the code in this example returns any markup content, which can make it difficult to tell what the program is doing without extensive documentation.

Extrapolating this small example, you may see why a framework is a more effective solution – otherwise, the time taken to implement each part of the application may be significantly increased and become exponentially more difficult to develop atop as the codebase grows.

The second code snippet is an example of a very ‘imperative’ approach. React and similar frameworks offer a more ‘declarative’ approach that allows developers to describe what a program should do, rather than how to perform the task. The following section will go further into the implications of these paradigms and the scenarios they are best suited for.

4.1.2. Declarative VS Imperative

Imperative programming can be described as giving direct, sequential instructions to the program to inform how it should behave. Alternatively, declarative programming allows developers to describe the output of their program without being concerned with the process of producing it^{xxiii}. Declarative code is often sought after in user interface programming, as it allows for a more human mental model of the program – a frequent example is that declarative code is like asking for a “table for two” at a restaurant, as opposed to imperatively telling the waiter exact instructions for how to find the correct table.

Declarative statements are more flexible by allowing the program to decide the control flow, reducing the surface area for bugs. HTML is an example of a purely declarative language – the page content is simply described using HTML syntax. No instructions for how to draw the content to the page are required.

To allow declarative functionality, React provides a ‘declarative API’ (Application Programming Interface) that handles the imperative operations involved with DOM interactions^{xxiv}. An example of this is the expression `{renderInvSlots()}` inside of the component’s markup. Under the hood, declarative languages and APIs all perform imperative operations at some point – computer processors operate by performing imperative operations. The goal of declarative programming is to allow developers a more human-oriented model for thinking about their program and how it behaves.

4.1.3. Functional Programming in React

The Functional Programming paradigm is a form of declarative programming, where a problem is broken down into a set of functions that map inputs to outputs, rather than a sequence of steps that inform the state of the program. While JavaScript and React are not strictly functional languages, there are several concepts that React borrows from functional programming.





A common theme in React's design and implementation is that *"UI/View is a function of state"*^{xxxv}, which informs the idea of 'pure' components. Pure components will always produce the same output for all input. A component's input is made up of:

- 'props', the arguments passed to a function component,
- 'context', data made accessible to a component via a component higher in the tree,
- 'hooks', functions available to all components, allowing access to the React's internal memory (e.g useState).

The component should not cause side effects (modifying variables outside its scope) and should not mutate input values (props)^{xxvi}.

4.2. Node.JS Overview

To allow players to collaborate and share a table, a back-end server is required to manage player data and synchronize players on the same table. To write the server-side code, I decided to use Node.JS.

Node.JS is a runtime environment for running JavaScript programs outside of web browsers^{xxvii}. While JavaScript was originally intended only for web browsers, major improvements in the language's features and performance (regarding Google Chrome's V8 Engine) proved its viability for server-side programs. The ability to write the back-end and front-end with the same core language provides an easier experience communicating between both sides, largely due to JavaScript's native support for JSON (JavaScript Object Notation).

However, JavaScript is often criticised as a back-end solution due to its roots in front end web development^{xxviii}. To make scripting and interacting with the DOM easier, there are several shortcuts and approximations that JavaScript allows – for example: 'truthy' and 'falsy' grouping of values in a Boolean comparison, allowing 'undefined' properties and variables, and the automatic type conversion with the equality (==) operator (leading to evaluations like `"22" == 22 = true`), just to name a few. The lack of type safety in JavaScript led to the development of a type-safe sister language called TypeScript, which is often upheld as a much more rigid way to write web applications^{xxix}.

Due to time restrictions however, it is unlikely that the project will use TypeScript, but converting the project to be type-safe with TypeScript will be the goal for future development, discussed in the "Future Work" section.

4.3. MongoDB Overview

To keep users from losing their progress, the data for each table must be stored in a database. I decided to use MongoDB for the database, with the back-end server making queries to it to serve and update user data^{xxx}.





MongoDB is a uses a document-oriented database model, storing BSON documents (a JSON-like data format) in collections. This is an effective way of storing unstructured data compared to a relational database, like a SQL database. SQL databases store structured data in tables, often connected using alphanumeric IDs called Primary Keys and Foreign Keys. Each record must have the exact fields for the table, making it a popular solution for user accounts and authentication. Storing files or documents is less effective however, often being limited to fields containing file paths.

As Dungeon Desk does not require any authentication to use, and due to the MongoDB support for JSON, I decided that it would be a more effective solution for the app prototype.





5. System Design and Considerations

As mentioned, the application consists of three distinct parts that communicate with each other: the client, the server, and the database. The client performs the majority of the state-related data operations, e.g. moving items between slots – this data is then sent to the server and relayed to the other clients connected to the table. After each operation, the database is updated.

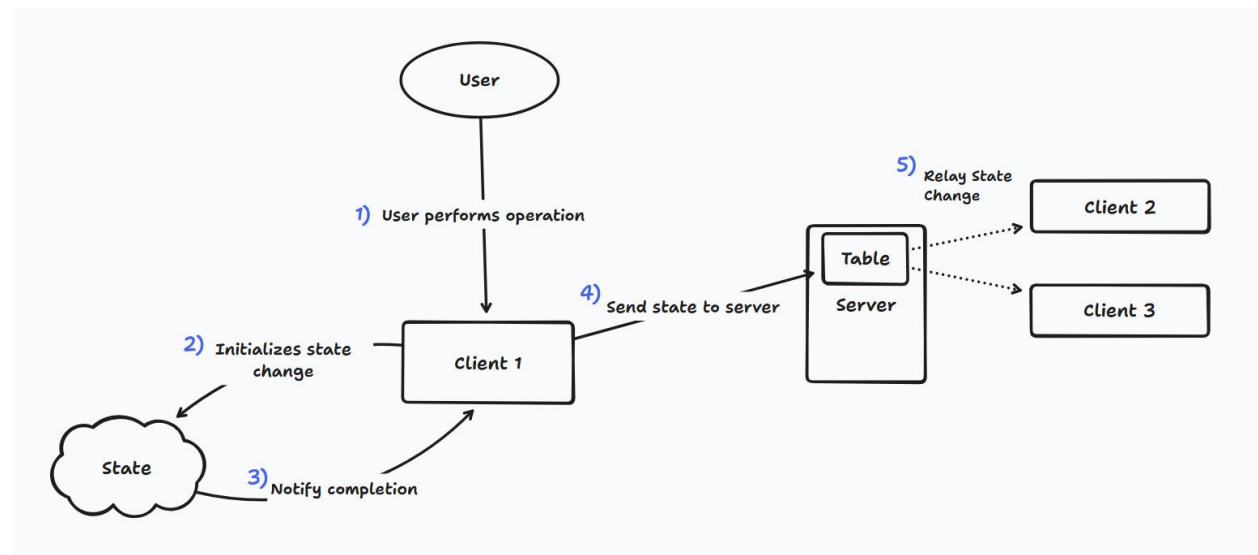


Figure 6: A diagram representing the process flow of the client-server communication.

The interactive client-side application can be described as a SPA (Single Page Application), and as a result, is completely decoupled from the back-end server. Some server-side frameworks and languages allow the server to send HTML to the client to update the view from the back-end (e.g. PHP, Django) – there exists framework for React called Next.js that allows for React components to be streamed from the server (named SSR or Server-Side-Rendering), resulting in faster initial load times for pages with a complex component tree^{xxxi}. It also allows for easier routing (using the URL to access specific parts of the application), enabling better SEO (Search Engine Optimisation). Despite these benefits, I decided to keep the front and back end separate – this allows for a much simpler mental model of the project and an easier experience debugging.





5.1. The client-side

In this system, most of the data processing is performed by the client. The user interface provides a means of allowing the user to alter the state of the table. As the state is altered, the server is sent updates, which are relayed to other users.

Therefore, it's important that that users can effectively manage and control this data. The user interface must accurately reflect the data and provide an intuitive way to update it and change it based on the actions the user takes. Mapping user interactions to changes in data presents several challenges that are covered in this section.

5.1.1. Visual Design

As mentioned, the layout of this Inventory style is heavily influenced by the standards of popular video games – specifically, games that have a survival aspect to the gameplay.

Inventories are often displayed as a grid, with each grid representing a place for a unique item. Items of the same type can often be ‘stacked’, meaning that multiple instances of an item can occupy only one slot. Players can drag these items from one slot to another to organise their inventory grid to suit their liking.



Figure 7: The popular video game Minecraft is an example of the grid-based inventory system.

The goal of Dungeon Desk was to echo this convention to allow players to intuitively manage their inventory with a fun and interactive interface. The styling of these individual elements went through numerous changes throughout the project, but the core layout remained consistent throughout the development process.

When considering how to style the page and components, my objective was to balance the conventional minimalism in modern web development with the experimental and thematic design of many video games. Successfully meeting the user standards requires a design that can invoke the same immersive feeling often present in the heavily styled user interfaces of video games, to allow the users to feel that the inventory manager is a part of their game^{xxxii}. Yet, I wanted to avoid an abundance of ‘visual noise’ in the user interface to keep the focus on the important page elements. I also wanted to ensure that first-time users would not need extra guidance when using the application, so I aimed to stick to modern conventions and keep the layout minimal.





Figure 8: A part of the user interface from *Elden Ring* - a popular fantasy video game. This is an example how user interfaces can follow the thematic style of a video game setting.

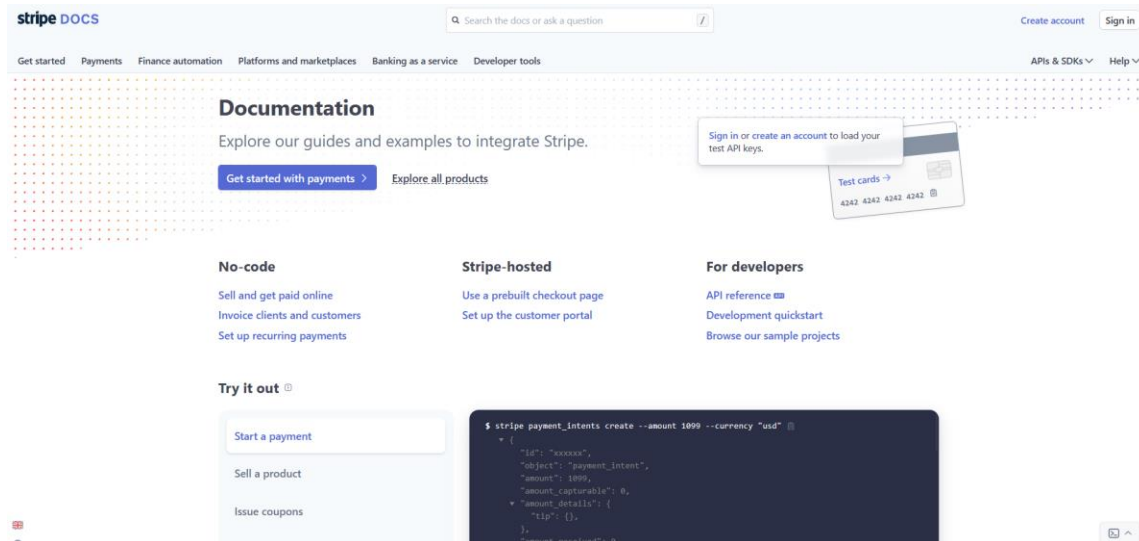


Figure 9: The documentation page for the Stripe payment service – a prime example for clean, minimalistic web design.





5.1.2. Component Hierarchy

As mentioned, React operates on the idea of components. A HTML document can be represented as a tree of elements and thus, A React user interface can be considered as a tree of components. It's important to consider the structure of the hierarchy of components, as this determines how data will flow through them and their relationships to other components in the tree.

A component is a child of another component if the 'parent' component renders an instance of the child component. The parent can pass down props to a child component – these can be state variables declared in the parent component or passed down from higher in the tree.

With the ability to pass down props, parent components have lots of control over the data the child component can access and – like HTML elements, a child component's position on the page will be relative to its parent, depending on its display and position properties.

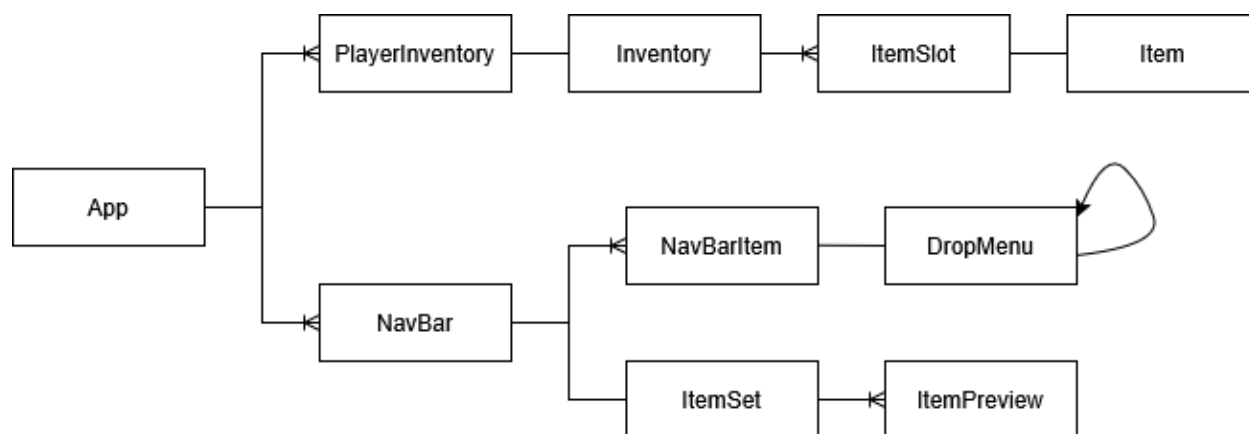


Figure 10: A graph showing the component hierarchy of the front-end user interface.

Components are often organised in a way that makes sense both for the data flow and the appearance. In earlier revisions of the application, the Inventory component housed the state data for each `<ItemSlot>`. When an `<ItemSlot>` needed to handle an event that is supposed to change the program state, it called a callback function passed down from its parent `<Inventory>` component.



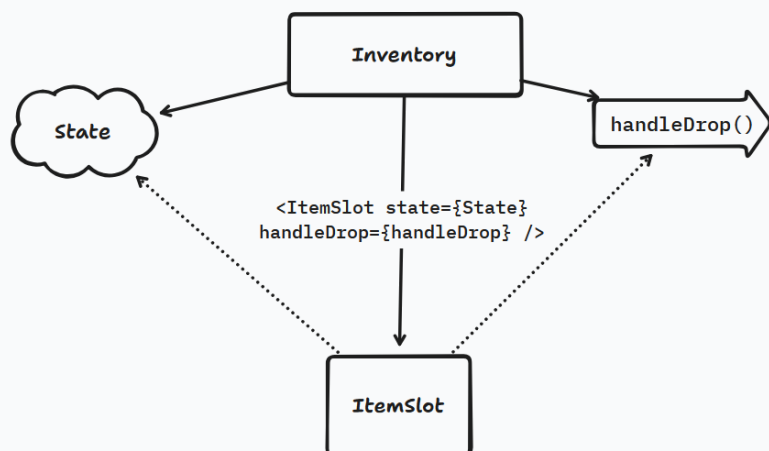


Figure 11: Process flow for an interaction between parent and child components.

This is a relatively simple example of managing application state, but it quickly became obsolete. For example – in the application, a new instance of an `<Item>` is made by dragging the relevant `<ItemPreview>` component on to an empty `<ItemSlot>`. The `<ItemPreview>` components are children of the `<ItemSet>` component, on the opposite side of the component tree. How would the `<ItemPreview>` perform the operations necessary to put an item in the slot? How would the `<ItemSlot>` component know that an `<ItemPreview>` was dropped on to it?

In React, if a component needs to influence the state of another, that state must be lifted to the lowest common ancestor^{xxxiii}. This means that here, the relevant state must be managed by the top level `<App>` component. Section 7.3 goes further in detail how this influenced the development of the application, and the tools React provides to manage it effectively.

5.1.3. State as Data

When representing items as components and data, it's important to consider what different item types are and how to uniquely identify them. In the state, an item's identity is primarily made up of its `InstanceID` and `ItemID`. The `InstanceID` refers to the unique instance of the item inside a player's inventory or container, generated each time an item is added to an inventory. The `ItemID` refers to the unique type of item, referring to an item in the `ItemSet`. The `ItemSet` houses all of the unique items that have been made by players.





In theory, this allows an `<Item>` component to simply take its `InstanceID`, `ItemID` and `SlotNo` as props, and render itself using the `ItemSet`, as long as it can access to all of its relevant data. Despite this, `<Item>` instances still hold all of their properties – this is to allow users to add unique properties to specific instances of an item.

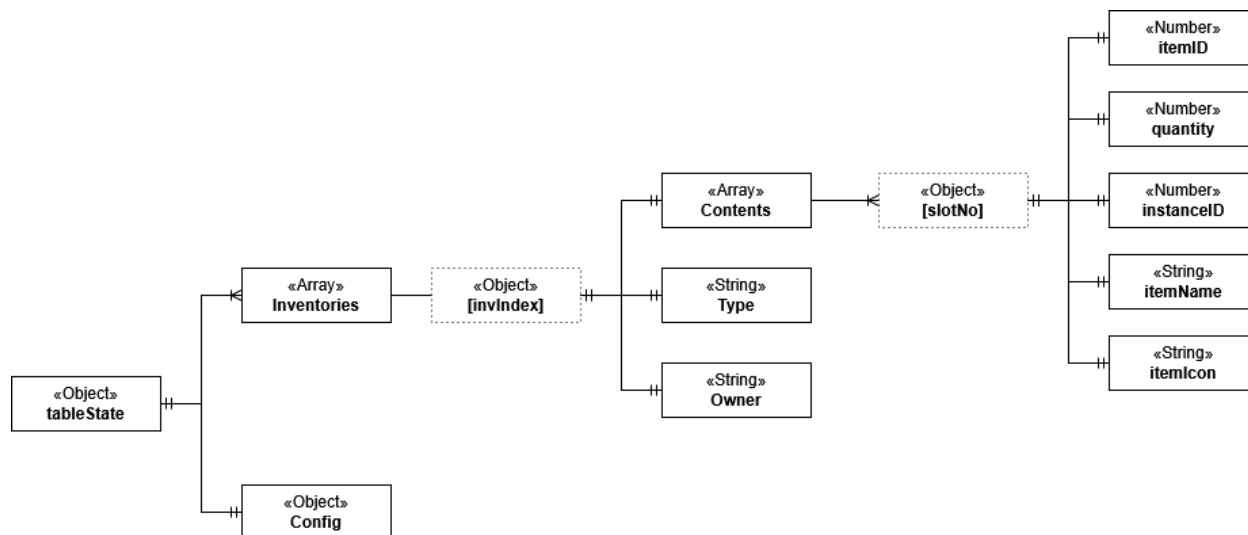


Figure 12: The JSON schema for the state store for a specific table.

Providing unique `InstanceIDs` for each item allows for item-stacking (storing multiple of the same item in the same slot with a number depicting its quantity). This is a quality-of-life feature in many games, especially as players often want to store a large quantity of the same item.

To associate items with their locations in an inventory, I decided to use a ‘sparse array’-like data structure, with each `<Item>`’s key representing the location of its `<ItemSlot>` component. A sparse array often refers to an instance of an array that holds no data in several locations. JavaScript objects can be used to implement a similar functionality:

```
sparse_arr = {  
  1: 'Element 1',  
  3: 'Element 3',  
  5: 'Element 5'  
}
```

Figure 13: A JSON implementation of a ‘sparse array’

In the code, the key associated with each item object referred to as `slotNo` – this represents the ID of the `<ItemSlot>` component holding the item.





When rendering the front-end, the `<Inventory>` component defines a range of numbers based on the current page – e.g: `{1...30}`, `{31...60}`. For each number in the array, it checks the contents of the inventory data for a key that matches the number. If there is a match, the `<ItemSlot>` is rendered with its corresponding `<Item>`, otherwise it is rendered as empty.

This way, rendering a page of `<ItemSlot>` components can be performed in $O(1)$ time complexity (constant time). The inventory always renders 30 `<ItemSlot>` components at a time (unless performing a search), and thus, the time taken to render an inventory is mostly independent to the number of items stored in the player's Inventory.





5.2. The Server-side

The following chapter describes all the tasks performed by the server, and how the server is set-up to serve the clients.

5.2.1. Real-time synchronization

Many web applications make HTTP requests to a REST API on the web to provide updates to a client view. REST APIs allow for communication between a client and server using HTTP requests (GET, PUT, POST, etc.), where the client requests the data, and the server responds. This is ideal for simple CRUD operations (create, read, update, delete), but when the application requires updates in real-time, there is a much better solution: web sockets.

REST APIs mostly use HTTP requests, which are inherently stateless and unidirectional. This means that a client must communicate with the server first before getting a response. To implement real-time updates this way, a client would have to poll the web server constantly, creating lots of traffic, latency and redundancy. On the other hand, web sockets are stateful and bi-directional, meaning that after a connection has been established between the client and server, the server can send updates to the client without the client having to send a request first^{xxxiv}. The latency of these updates is also reduced – after a connection has been established through a web socket, data can be sent without performing connection handshakes (initial communication protocol between the client and server) which significantly reduces the round-trip-time for each update. Alternatively, HTTP requests require a new handshake for every exchange^{xxxv}.

5.2.2. Server State Management

As mentioned, the client application is responsible for most of the state changes. The server mostly echoes these changes to other clients. One important feature of the system performed by the web server though, is the ability to reverse changes (the ‘undo’ feature).

This is a popular feature in software for manipulating data through a user interface, as users may often make accidental changes to their data that they did not intend to. Many methods exist for implementing this, with varying degrees of efficiency and complexity^{xxxvi}.

In a scenario where multiple people updating the same data, it’s likely that a user will want to revert their own changes at some point. Using an undo system that stores the whole state of the application incrementally means that when one user attempts to undo their change, any other succeeding changes by other users are discarded. A single user could easily discard several of other user’s changes by accident. Storing multiple instances of the whole state for a table is also very memory intensive.

For a better solution, the program can instead use the difference between two state objects to manage undo operations. For this, there exists a library called ‘recursive-diff’ for JavaScript^{xxxvii}. The library provides two main functions: `getDiff()` and `applyDiff()`.





Running `getDiff(oldState, newState)` will return an object representing the changes from the old state to the new state. The function `applyDiff` is also provided, which can take the result from the `getDiff()` function and apply it to an object.

Mathematically, if there exists a state change $S \rightarrow S'$ such that S = current state, and S' = next state:

$$f(S', S) \rightarrow d, \text{ where } f(x, y) : \text{getDiff}(x, y),$$

$$g(d, S') \rightarrow S, \text{ where } g(x, y) : \text{applyDiff}(x, y).$$

Implementing this, for each state operation the program calculates two values: Forward difference (o1) and backward-difference (o2). The forward difference refers to the result of taking the difference from the current state to the new state. This will give us an object that can be applied to the state to reverse the changes made. The backward-difference (o2) is the result of reversing this order, taking the difference from the new state to the current state. Likewise, applying the object to the current state will produce the new state. This becomes important very soon.

The server will store these values in an object and push them to the table's 'undo' array, which stores all these differences, alongside the respective clientIDs. When the client initiates an undo, server locates the operation, pops it from the array and applies the backward-difference to the current state. This system should allow users to undo their own changes without reversing others – as long as other clients have not overwritten those changes by tampering with the same items.

What if another user does overwrite that change, though? For example: One player may rename an item. Later, a different user renames the same item. Now, if the previous user undoes their rename operation, the last user to rename the item loses their change as well.

If this undo operation is conflict-free, then applying the forward-difference of the state change to the current state should result in an object that is identical to the current state.

$$f(S, S') \rightarrow e,$$

$$g(e, S') \rightarrow S' : \text{if the undo operation is conflict-free.}$$

If `applyDiff(e, S')` is the same as the current state, then the program can call `applyDiff(d, S')` to perform the undo operation. If the result is not identical the current state, there is a conflict (overwritten changes) and thus, the operation can be discarded.

Performing the extra checks and calculations can be quite computationally expensive in regards to time. Equality-testing two JavaScript objects (by converting them to strings) takes linear time. Getting the difference between two objects takes $O(K^M)$, where K is the average number of keys at each level and M is the depth of the object^{xxxviii}. This function is ran twice at each change in state.





This can be run asynchronously as to not cause slow downs in performance, but this may lead to race conditions.

6. Implementation

The following sections will detail the process of writing the software to accomplish these requirements, and any design decisions that were made or altered along the way.

6.1. Inventory Features

As the purpose of the Inventory component is to allow users to manage their inventory, it must allow them to add items, move items and organise them as they desire. An `<Inventory>` component displays 30 `<ItemSlot>` components. These components can either be empty or render a single `Item` component.

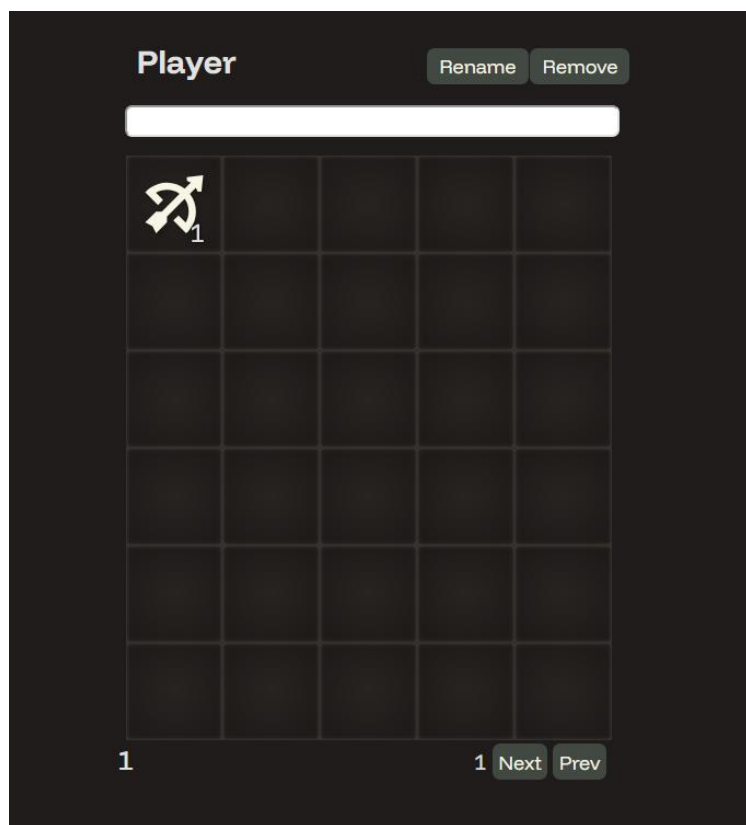


Figure 14: The current inventory layout and design.

To allow players to interact with their inventory, drag-and-drop functionality was necessary. Originally, I attempted to create my own drag-and-drop implementation for the application using vanilla JavaScript, but this quickly became time-consuming and presented problems, especially





related to the hierarchy of components. It became clear that using a library was a much better option, for which I decided to use 'dnd-kit'.

To specify a Draggable component, Dnd-kit provides the `useDraggable()` hook. Calling `useDraggable()` returns four variables – one of which is `setNodeRef`, a React Reference for specifying the JSX element to be affected by the drag functionality. The program may also pass an object as an argument to the hook, specifying a unique ID for the element and the data to be accessed through an `OnDragEnd` event. This will allow us to identify the item instance in the event handler for performing program logic.

Similarly, a drop target can be specified with the `useDroppable()` hook, which also provides a reference to be attached to the desired JSX element and takes an object as an argument for specifying the data accessible to the event handler. Attaching this to an `<ItemSlot>`, the program can specify the ID of the slot and the types of items accepted by the slot.

The components that use these hooks must be wrapped in a `<DndContext>` component. The `<DndContext>` component allows us to pass an event handler as a prop to `OnDragEnd`. A caveat to this is that it heavily encourages us to define the handler's behaviour at the level where the context is defined – in this instance, the top-level `<App>` component. This revokes our access to some component-specific operations, but the flat state structure made this relatively trivial.

6.2. Visual Design

The appearance and organisation of the front-end components were iterated upon several times throughout the project. The first prototypes of the app utilised a minimalistic slate-grey colour palette, where there is a strong contrast between the inventory container and the item slots. The icons were also coloured and shaded.

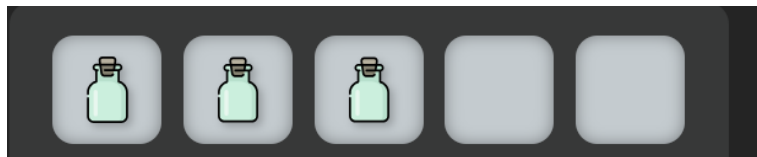


Figure 15: A screenshot of the original inventory design.

In user testing, some users expressed that a different colour scheme may be better suited for the application. The cool-toned slate grey and rounded minimalist style did not pair well with the conventional fantasy atmosphere of TTRPGs.

The colour scheme was altered to be warmer and softer to provide a nicer atmosphere. The item slots have much less contrast and stand out far less from the background. Icons are also monochrome and cream-colored, making them the focal point of the `<ItemSlot>`, while the dimmer backgrounds of each slot reduce the visual noise.



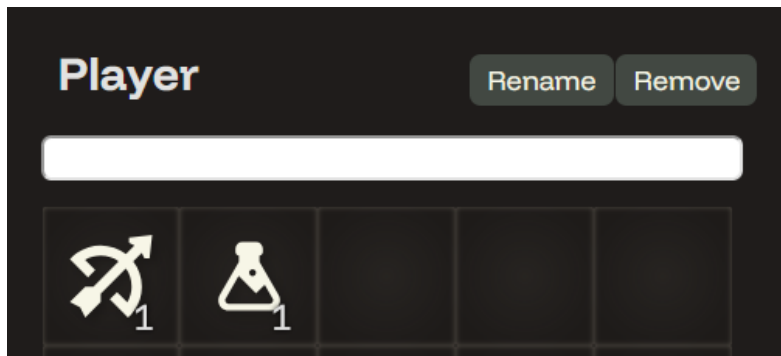


Figure 16: The updated inventory design, with better styling.

Upon user testing, all users expressed that they felt this was an improvement on the previous design and would blend in better with their games – a design goal stated in the user requirements.

Another aspect of the design that changed was the `<ItemSet>` component. In early prototypes, the component was visible through a hidden sidebar. A translucent arrow projected out from the right side of the screen that would reveal the component when hovered.

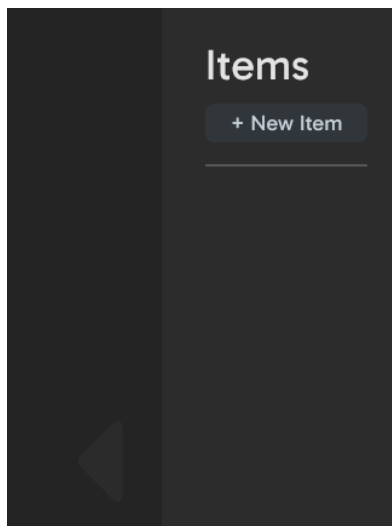


Figure 17: A screenshot of the early prototype `<ItemSet>` component.

However, testing feedback proved that this was ineffective, for several reasons:

1. The arrow is difficult to see and has very little indication of its purpose.
2. The component would often obstruct important page elements on narrow displays and could easily be revealed by accident.
3. Inaccessible to use on mobile devices – users had trouble keeping the menu visible due to the absent 'hover' functionality on phones and tablets.





Based on this feedback, I decided to abstract the `<ItemSet>` component and place it inside a ‘hamburger menu’ as opposed to the hidden side bar. In user interfaces, a ‘hamburger menu’ is a design convention for implementing a toggle-able list of actions, especially on mobile devices. It is represented by three horizontal lines and often occupies a spot on an application’s nav-bar. In this instance, the icon sits to the right of the nav-bar, as to not be coupled with the other nav-bar options.

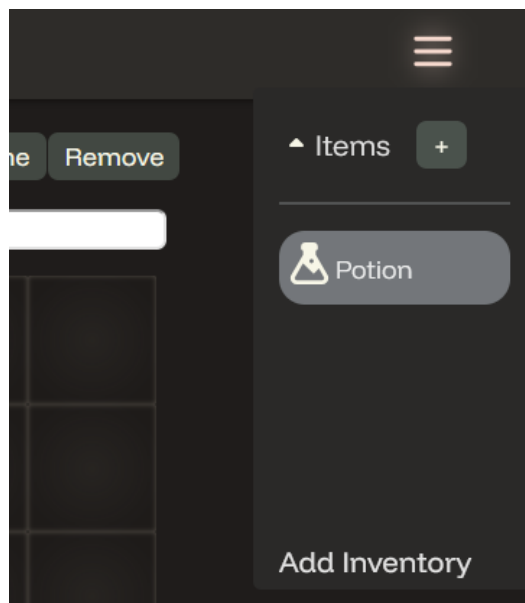


Figure 18: A screenshot of the improved `<ItemSet>` component.

This provides a much clearer means of accessing the `<ItemSet>` component and is equally as effective on mobile devices and computers.

6.3. Front-end state management

For managing stateful variables, the most common method is the `useState()` hook. When `useState()` is called inside a component, it returns a variable referencing the state value, and a setter function for changing the value. These can be passed as props to child components.

In the example from section 5.1.3, I mentioned how the state of the inventory would need to be stored in the `<App>` component to allow the two components to interact. To allow the `<ItemSlot>` and `<ItemPreview>` components to access the required data, this data would have to be passed through many component layers. To get to an `<ItemSlot>`, it must be passed from `<App>`, to `<PlayerInventory>`, to `<Inventory>` and to `<ItemSlot>`. It would then also have to be passed all the way on the other side of the tree for the `<ItemPreview>` to access it. This is known as ‘prop-drilling’, and it often leads to lots of unnecessary repetition.





Instead of passing data as props, React allows for ‘context’ – this specifies data that should be available to all descendants of a component. This is implemented with the `useContext()` hook and the `<Context.Provider>` wrapper^{xxxix}.

In this application, the table’s state data is managed by the top-level `<App>` component and is passed as context to all child components. In code, the program could store the table state with `useState()` in `<App>`, then pass the state variable down to the child components that must read it. This allows each component to read each other’s state, but they still cannot interact with each other.

While the setter could be passed to the context, this would tightly couple the components together, causing an abstraction leak. A better solution is to define discrete actions that update the data in a specific way. For this, React provides a reducer, via the `useReducer()` hook.

A reducer is a pure function that takes an old state value and an operation to be performed and returns a new state value^{xl}. This is a very popular choice for managing complex and deeply nested state and is what I chose to use for state management.

A reducer allows us to define a finite number of operations that can be performed on the data, which prevents code repetition and reduces the likelihood of error. A flat state structure is also deeply beneficial for this project, as for the multiplayer system to work, the top-level component must send the state data to the server.

The `useReducer()` hook provides a dispatch function to be called to update the state. This dispatch function takes an object with one required property ‘type’, which specifies the action to be taken. Any arguments to be passed to the action can be passed to the object.

For instance, the `MOVE_ITEM` action takes the item to be moved, the inventory it belongs to, and the new location, to update the data so that the state data stores the item in the new location. The reducer itself is responsible for checking that the operation is valid, for example, the new slot location is not already taken by another item. These actions can be reused across the application, so if there is a change in the structure of the data that breaks the state change, there is only one part of the application that must be fixed.





6.4. Feature Requirements

This section goes in-to depth on the implementation of some of the specific features outlined in the user requirements.

6.4.1. The Item Set

As shown above, the Item Set is a list of items that have been made by users on the table. All items made on the table should appear here, as to allow multiple players to have their own instances of the same item as other players. Any users connected to the table can drag an item from the list of items and place it in any `<ItemSlot>` on the table.

The items are listed as `<ItemPreview>` components that have their own drag-and-drop properties. An `<ItemSlot>` can distinguish between this component and an item component, as they must be treated differently.

When an `<ItemPreview>` is dropped on to a slot, the `handleDropEnd` event handler is called from the `<App>` component. The reducer's `ADD_ITEM` action is called – this checks the details of the call to ensure that the operation is valid (e.g. ensuring the item is dropped on an empty slot), then updates the state of the application.

6.4.2. Inventory Pages

Inventory pages, as specified in the user requirements, allows the user a great deal of flexibility when storing items. The `<Inventory>` component displays 30 `<ItemSlot>` components at a time, by mapping an array of integers `[1...30]` to the `renderItemSlots()` function. To render a specific page, I used a state variable storing the current page (e.g. page 2), using the page number to generate the mapped integer array with a simple function:

```
S => (pageNo - 1) * 30 // S is the starting slotNo
```

```
E => S + 29
```

Then using `Array.fill()` to generate a list of integers from S to E, mapping each number to an `<ItemSlot>`, passing the number into the `slotNo` prop. (This is used by the component to render the item and handle events).

The current page number for an inventory is not stored in the table state store. State information in `tableState` is shared amongst players, so state specific to each client view should be kept separate.

6.4.3. Creation of tables

Users visiting the root page at `app.tayjordan.site` are greeted to a welcome screen. They can click “Create a table”, enter the name of the table, then click the ‘Create’ button. On a working





application, this will give them a table ID and a link for their table, that they can visit to access the table data. Anyone with the link can also access the table and update it.

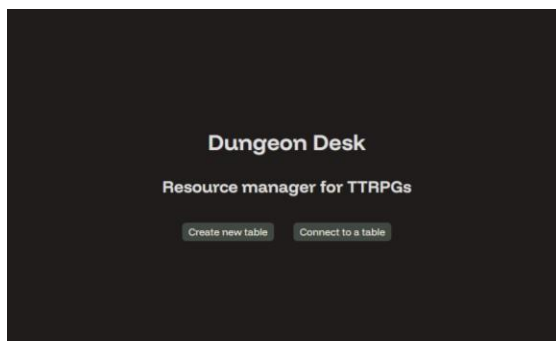


Figure 19: The homepage of the application.

To allow access to specific parts of the application using a URL, the 'React router' library is used. A user can skip the welcome screen by navigating to 'site/newTable' for the table creator, or 'site/app/?tID=[tableID]' to access a specific table. This is slightly unconventional – the URL for a web page is often a directory to a specific document on the server. For example, a portfolio website may have an 'About Me' page at 'site/about_me.html'. React apps are built from components rather than documents, so managing URLs can be tricky.

The top-level component in the application is `<AppRouter>` -- here, the components act as separate web pages, using the URL to decide what is visible on the page.

```
export function AppRouter() {  
  return (  
    <BrowserRouter>  
      <Routes>  
        <Route path="/newTable" element={<GenerateTable />} />  
        <Route path="/" element={<Home />} />  
        <Route path="/app" element={<App />} />  
      </Routes>  
    </BrowserRouter>  
  )  
}
```

The `<App>` component at the top level of the design diagram is a child of `<AppRouter>`, to separate the routing from the main functionality of the app.

The `<App>` component at the top level of the design diagram is a child of `<AppRouter>`, to separate the routing from the main functionality of the app.





When the user attempts to create the table, a GET request to the back-end sever is made; “/createTable_[tableName]”. If the table is successfully created, the server responds with a JSON string formatted as: {type: SUCCESS, tableID: [tableID]}. Otherwise, a response is sent with {type: ERR}. The front-end displays an error message if the server returns an error message, or if the request times out without a response.

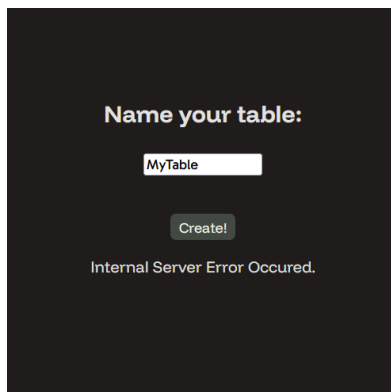


Figure 20: An error shown on the table creator page.

While waiting for the response, an animation of a spinning D20 (20-sided die) is displayed.

On a successful creation, a new blank table is stored in the database. The ID is generated from concatenating a random string with the base64 encoded table name.

6.4.4. Real-time synchronization

As mentioned in section 5, Web Sockets are used to enable the real-time functionality. With web sockets, a client initiates a connection to the server. Once this connection is made, the client and server can communicate bi-directionally, meaning the server can send updates to the client without the client having to send a request.

When a user navigates to their table using a URL, the table ID is collected from the GET request, with React Router’s useSearchParams() hook. The client then initiates a web socket connection to server, by using a URL. The URL would look something like ‘wss:[site]/ws/[tableID]’. The protocol identifier ‘wss’ is used to specify a secure web socket connection rather than a HTTPS request.

On the back-end server, to allow for web socket connections, a reverse proxy must be configured for the /ws/ route. This means that any incoming connection at the /ws/ location is re-directed towards the port for web socket connections – creating a buffer between the server port and the client for more security and allowing for a more readable URL. This is configured using Caddy.

On the front-end, the application listens for any changes in the table state variable using the useEffect() hook. The useEffect() hook allows us to perform side effects – it takes a function and a list of dependencies as arguments. When a state variable changes in the dependency array, the function is called.





This is used to update the server whenever a client user updates the state of their table. In the `<App>` component, a `useEffect()` is defined as follows:

```
useEffect( () => {  
    sendState(tableState);  
}, [tableState])
```

So now, the server relays this to all other clients on the same table.

When a client application updates the table state, the server will send a message with the current state to all connected clients. This is handled by a `manageSocket` object stored in a `Ref` in the `App` component. (A `ref` is a component-bound variable that persists between re-renders.)

The socket manager is initialised with a `useEffect()` hook, with an empty dependency array. This specifies that the passed function should run when a component is first mounted on the page.

The socket manager takes a callback function from the `App` component – this function updates the table state by passing a new object to the reducer. When the socket manager receives an incoming message from the server, it will call this function to update the state of the client.

Following the process flow, you may see a problem – when the client updates the state in response to a message on the server, the `useEffect()` hook will detect a change in the state of the table, and thus, relay this state back to the server. The server then takes this state and relays it to the other clients – all of which perform the same cycle. This traps the system in an infinite loop.

The solution that I implemented was to use a variable `receiving` in the `<App>` component to signal when data is being received from the server. The setter function for this variable is passed to the socket manager, setting `receiving` to `true` upon receiving a message from the server.

Now, the `useEffect()` for updating state must check to see if `'receiving'` is set to `true`. If it is, the state is not sent to the server – instead, the variable is set to `false` to indicate that the client is no longer receiving data.

While no problems have been encountered during the user testing of this feature, I would like to implement a solution that is a lot more robust in the future. In React, the `useEffect()` hook is often regarded as a 'last resort' for triggering side effects, as overusing it can be harmful to the stability of the application. A better solution would be to send the state data in a function that is intentionally called when a *user* updates the state – not as a side effect of the state variable changing.





6.4.5. Undoing Changes

The undo feature implements the methodology outlined in Section 5.2.3.

When a client connected to a table sends a state update, the forward difference and backward-difference is calculated from the current state and incoming state. This is then pushed to the table's 'undo array' – for indexing purposes, this is a key-value JavaScript object. Each client object keeps an array containing the indexes of their undo operations in the table array – this allows the client to identify their own operations from others without keeping a redundant copy. When a client undoes a change by clicking the nav-bar icon, a message is sent to the server with the 'undo' type.

For testing the conflict handling, I would simulate multiple players by connecting on multiple internet browsers. Client 1 would rename or move an item, then Client 2 would move or rename the same item. In these instances, if Client 1 tried to undo the change, nothing would happen. This is close to the intended functionality – Ideally, a message would be displayed on the client screen informing them of the problem. Unfortunately, I could not implement this before the deadline.

For a different test, Client 1 and Client 2 would perform operations that should not cause conflicts, by changing the properties of different items. This seemed to work fine.

To stress test the feature, I performed a test where a client would perform several operations and attempt to undo them all. After the undo button was clicked several times, an odd bug occurred where the components representing item instance would duplicate – there would be multiple components in multiple slots representing the same item. In the future, I would like to iron this feature out – but currently, it is quite unstable.

6.4.6. Sub-Inventories

In TTRPGs, players are often given the option to store items in containers – like bags, pouches, or chests to allow players more ways to organise their items. This feature is often implemented in video games, with the containers visualised as smaller inventories that take up less screen space.



Figure 21: An example of a container in Baldurs Gate 3^{li}

To implement this in the application, some the aspects <Inventory> component would need to be reworked slightly. Before the implementation of sub-inventories, the component was tightly





coupled to the structure of player inventories. This means that inventories would only render correctly with the current state structure.

Instead, the `<Inventory>` component can be abstracted – instead of assuming that the inventory belongs to a player, the functionality is reduced down to simply taking an inventory’s contents as input and rendering them to the page.

For functionality specific to players, the `<PlayerInventory>` component takes the data for a player’s inventory and uses it to render an `<Inventory>` component with the contents. It also provides the correct styling for the page and adds additional features.

In *Dungeon Desk*, a container is a type of item, as it can be held in an `<ItemSlot>`. When an item is created with the “Container” item type, a new inventory is added to the array of inventories. The item gains additional properties – one of which references the location of the inventory data.

To implement the visual part, I created a `<ContainerInventory>` component that locates the inventory in the state data and renders an `<Inventory>` component. This component is also specifically written to be displayed inside of a Modal Box, so that the user can easily toggle its visibility.

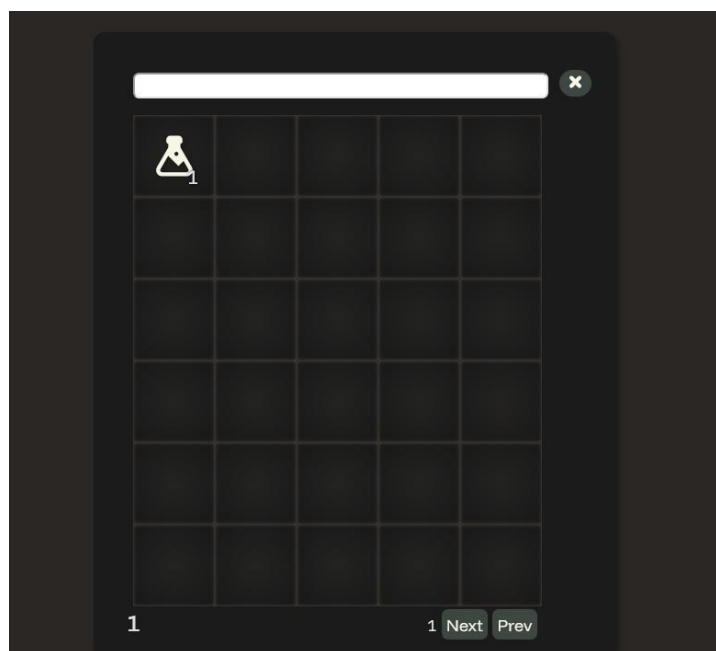


Figure 22: A `<ContainerInventory>` component, displayed inside of modal box that can be toggled.



6.5. Mobile Adaptations and Considerations

One of the user requirements for the application was to ensure that it is sufficiently mobile-friendly. This creates two main objectives for developers:

- 1) Ensure that all web page interactions can be easily performed with both mouse/keyboard input and touchscreen devices,
- 2) Ensure that all necessary page content is optimised for mobile screens: That is, narrow aspect ratios (especially portrait screens), and a smaller area to display content.

Some JavaScript event handlers are designed to serve both 'Mouse' and 'Touch' events, but some interactions must be implemented separately for both types of devices.

This was an important consideration when implementing the Drag-and-Drop functionality of the program – many React libraries do not have sufficient touchscreen support. The library 'dnd-kit' provides 'sensors' to specify which devices can use the drag-and-drop functionality.

In regard to the visual design, some CSS properties provide sufficient results on all displays, but in many scenarios, changes must be made to the CSS properties for different device types. For this, we can use Media Queries.

A Media Query specifies a block of code to include if a certain condition is met. In this scenario, if the width of the client device is below a certain threshold, before applying the properties in the block.

```
@media only screen and (max-width: 600px) {  
  .inventorySpace {  
    margin-left: 0;  
    margin-right: 1rem;  
    width: 22rem;  
  }  
}
```

Here, I specified the max-width of the inventorySpace container to be 22rem, if the max-width of the screen is 600px (that is, the screen width is smaller than 600px)





6.6. Web server set-up

The web application is hosted at `app.tayjordan.site`. It served by Caddy, an easily configurable web server with simple set-up for HTTPS and reverse-proxying. This caddy server runs on a DigitalOcean droplet, running Ubuntu as its operating system.

To serve the static site, the project is first built with `npm run build`. This makes optimizations and converts JSX (JavaScript XML) into regular HTML and JavaScript, for running in any browser. The Caddy configuration file points to the location of these files and serves them to the user at `app.tayjordan.site`.

For handling CRUD operations, namely: creating new tables and connecting to tables, the Node.js server uses Express.js.

To manage the real-time updates, the front-end must make a connection to the server's web socket, which operates on a different port. Here, a reverse-proxy is set up at `'app.tayjordan.site/ws'`, to take incoming requests and forward them to the web socket port, which allows for a buffer between the client and web socket server.





7. Conclusion and Future Work

To conclude, the final application successfully met most of the user requirements. As of writing, the finished product can be accessed through the URL <https://app.tayjordan.site>.

The user tests on the final application gave positive feedback, with several users expressing that they would prefer to use this tool over their previous methods. The results also reported that desktop users and mobile users were equally satisfied with the final product.

This app presents an example of how interactive software can help improve the experience of tasks that are unengaging by tailoring the user experience to be fun and intuitive. Many software applications that exist to accomplish seemingly similar goals as other software, but provide an experience tailored to different scenarios.

The rest of this chapter will detail the improvements I hope to make in the future, and what I had hoped to accomplish within the timeframe that was not accomplished.

7.1. User feedback results

The results of the user testing showed that the application met the user requirements to an adequate degree but still showed some dissatisfaction with the features available and some of the visual elements.

In two of the anonymous surveys, users expressed that some of the animations and dynamic elements could be improved – especially regarding the Drag-and-Drop functionality. One such user expressed that they would like to see a smoother transition after dropping an item to a slot.

Another user provided extra features that they would like to see – specifically, being able to assign yourself (as a client) to an inventory, and toggling the visibility of other inventories on the table.

Many of these features were planned but changed as the scope of the project become more concrete throughout development.

7.2. Future Work

While this project laid a great foundation for a project and gave me a great opportunity to expand my skillset, some aspects of the project were not given the care they needed during development. On both the front-end and back-end, I took some shortcuts and focused on having something that worked rather than something that was robust and maintainable.





7.2.1. Back-end

With very few concurrent users and a relatively small quantity of data to manage, the response times and latency for each operation is low, and the application feels responsive and quick. However, if the service becomes more popular and there are more concurrent users, the performance will start to quickly deteriorate, and the current architecture will not support a larger number of concurrent users.

The main cause for this will be the hardware – the server handling all operations is currently a single-core Ubuntu server with 1GB of RAM. Even with optimizations, the server will likely struggle to provide real-time updates with high levels of traffic, so migrating the server to more capable hardware would be necessary to support a large user-base.

Optimizations could be made to the back-end, however. While Node.js is fast, the server does not use multiple processes or threads, simply running all concurrent operations on the single core. In a multi-core system, I would opt for using ‘clusters’ or ‘worker threads’ in Node to take advantage of multiple cores^{xlii}.

With even larger user-bases, it may be worth switching to a different language in the back-end, due to the difficulty of utilizing multiple CPU cores in Node. Examples of these languages include Erlang and Elixir; these are functional languages that compile to BEAM bytecode and implement the Actor model for handling concurrency^{xliii}. They are known for underpinning popular messaging apps like Discord and WhatsApp, as well as a large portion of the telecommunications industry^{xliv}. Other languages like Go and Rust may also offer better performance and reliability for concurrency^{xlv}.

A more robust database solution may help the performance and reliability of the application in large-scale scenarios, like SQL or Cassandra, for better write-speeds. This would likely be very far in the future, primarily involving user authentication and other features^{xlvi}.

While the application does not store any personal data, the security of the application is quite unsafe. When performing real-time updates, there is currently no validation on the input data. This means that a single user could corrupt a whole table (and potentially the whole web server) by simply injecting the site with corrupt JSON through the web browser’s debugger. Before publishing a release of the application, sufficient validation must be performed on the web server to ensure that table data fits a specific schema. It is not recommended to rely heavily on the application in its current state.





7.2.2. Front-end

There are many front-end features that I had in the original user requirements, but did not make it into the app due to the time constraints. Namely, naming/favouriting pages, “combat tracker”, and other features.

Before going to implement other features, I would like to make some further refactors to the code base. The structure of some of the components could be better, and it’s possible that data structures used for state-management could also be improved or customised to make state stepping features (undo, redo) easier to implement.





Appendix

A1. Professional Considerations

The sections below will discuss the ethics of this project, and what must be considered to ensure that the project falls in line with the principles outlined in the BCS Code of Conduct.

A1.1 BCS Code of Conduct

During all parts of developing this project - including conducting research, data collection and application development, I will ensure that all activities fall in line with the requirements set out by the BCS Code of Conduct.

To ensure the privacy and security of others, all user testing will be conducted within strict ethical guidelines set out by the User Testing and Compliance Form. Any data collected will be completely anonymised and will be strictly related to the development of the application. There will be no danger or financial incentive involved in the testing process, and users must give their consent before participating, as required in 1(a).

References to any third-party content will be carefully considered to rule out any possible copyright infringement as required in 1(b).

Adequate accessibility is a key goal for the finished project, and so the project must be representative of a diverse group of people's needs and requirements. There will be no discrimination in the selection of user testing participants, and all of their needs and feedback will be considered equally, as required in 1(c), and 1(d). The application will feature no content that could be deemed harmful or offensive.

For this application to be of practical use, I will make sure that I am up to date with recent developments in web hosting and software engineering technology and stay aware of any potential security flaws in the technology that I use. No user-testing or public deployment will take place without first ensuring that the application abides by the relevant legislation.

A1.2. Ethics Considerations

There are certain ethical considerations that must be answered before parts of this project can be completed. The primary two considerations will be in regard to User testing, and Copyright infringement.





A1.2.1. User Testing

To carry out user testing of the software without needing further ethics approval from the university, the testing must abide by a set of safety and integrity guidelines set out in the Software Testing with Users Compliance form. The following section will discuss how these requirements will be met.





- 1) *Participants were not exposed to any risks greater than those encountered in their normal working life.*

The digital software poses no threat to users or other individuals and will not be developed to invoke any emotional or physical reaction from the user.

- 2) *The study materials are paper-based or comprise software running on standard hardware.*

The software being tested should be accessible via the participant's laptop or desktop computer, and runs in the browser.

In early stages of development, or when testing client-side components, the user may need to download a HTML web applet that runs in the browser. No one will need to download external libraries or frameworks to complete user testing.

- 3) *All participants must explicitly state that they agreed to take part, and that their data can be used in the project.*

An introduction script will be read, detailing exactly what data will be collected and how it will be stored. The data will be completely anonymous, with no names or personal details attached, and users will be given examples of tasks they may be asked to perform. Testers are also assured that they may withdraw from this study at any time. Users may ask any questions beforehand, before they can give explicit consent to take part in the study.

- 4) *No incentives will be offered to the participants.*

There will be no reward or incentive of any kind offered to the participants for partaking in the study. It will be ensured that testers may decide to take part entirely at their own will.

- 5) *No information about the evaluation or materials is intentionally withheld from the participants.*

In the introductory script, users are explicitly told the purpose of the study and exactly what data is collected, and how it will be used.

- 6) *No participant is under the age of 18.*

No participants under the age of 18 will be offered to take part in the study - potential participants are selected from a pool of students I know personally at the university, or attend another higher-education university in the country.

- 7) *No participant has a disability or impairment that may have limited their understanding or communication or capacity to consent.*

Participants with disabilities that may limit their ability to consent will not be offered to take part in the study.

- 8) *Neither I nor my supervisor are in a position of authority or influence over any of the participants.*

As a student, I hold no authority or influence over the participants of the study. While my supervisor may be deemed influential by some participants, no personal details of the participants will be shared with the supervisor, and any participants may choose to withdraw, even after the study is completed.

- 9) *All participants will be informed that they can withdraw at any time.*

This is stated within the introductory script, which will be read out before each study begins.





10) *All participants will be informed of my contact details, and the contact details of my supervisor.*

At the end of the study, participants are given my personal email address, and the email address of my supervisor.

11) The evaluation will be described in detail with all of the participants at the beginning of the session, and participants will be fully debriefed at the end of the session. All participants will be given the opportunity to ask questions at both the beginning and end of the session. All details of the evaluation, including examples of the tasks and data collected, are provided in the introductory script, and a recap of the evaluation is included in the debrief script. Participants may ask questions before the study begins, or after the study ends.

12) *All the data collected from the participants will be stored securely, and in an anonymous form.*

None of the participants' personal details will be stored in the data or logged, and the data will be stored locally on my password protected home machine.

A1.2.3. Rights of third-parties

Some of the user requirements described in the requirements analysis involve reference to third-party intellectual property. Providing quick access to rules and content is an objective that encompasses some of the non-functional requirements, but the rights and copyright guidelines of third parties must be carefully considered before any of their content is referenced.

Wizards of The Coast – the publishers of Dungeons and Dragons – permit use of all content described in the [Systems Reference Document 5.1](#) under a Creative Commons license (CC by 4.0), as stated on their website. Any content that references Dungeons and Dragons in the application must be stated on the SRD 5.1.

Other publishers like Paizo (Pathfinder), and Massif Press (Lancer), each have their own third-party licenses and reference documents for what may be published using their intellectual property.

While the project is designed to be a tool for any TTRPG system, many players will wish to skip tasks like making custom items or designing their own combat system. Any content that is referenced by the application must have explicit guidelines for use in non-commercial projects before it can be implemented.





A1.2.4. Signed User Testing and Compliance Form.

Software Testing With Users Compliance Form – UG & PGT Projects*

School of Engineering and Informatics, University of Sussex

This form should be used in conjunction with the document entitled “Ethical Review Guidance for Student Projects”.

Prior to conducting your project, you and your supervisor must discuss the ethical implications of your project. If your proposed project complies with **all** of the points in this form, then may use this form.

If you use this form, you must also prepare either (a) an information sheet and a consent for your participant to sign, or (b) prepare an Introduction and Debriding Script to read to your participant before the test (see appendix below). These must be approved by you supervisor.

Include the signed copy of this form with your final project report/dissertation.

If your proposed project does not comply with all the points, you should refer back to the “Research Ethics Guidance for UG and PGT Projects” document for further guidance.

-
1. Participants were not exposed to any risks greater than those encountered in their normal working life.

You have a responsibility to protect participants from physical, mental and emotional harm during the testing. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, physical hazards or discomfort, emotional distress, use of

*This form was originally developed by Professor Steven Brewster at the University of Glasgow, and modified by Dr Judith Good for use at the University of Sussex with his permission.





sensory deprivation (e.g. ear plugs or blindfolds), sensitive topics (e.g. sexual activity, drug use, political behaviour, ethnicity) or those which might induce discomfort, stress or anxiety (e.g. violent video games), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback.

2. The study materials are paper-based or comprises software running on standard hardware. *Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, mobile phones, and tablet computers is considered non-standard.*

3. All participants must explicitly state that they agreed to take part, and that their data can be used in the project. *Participants cannot take part in the test without their knowledge or consent (i.e. no covert observation). Covert observation, deception or withholding information are deemed to be high risk and require ethical approval through the relevant C-REC.*

If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, the data is to be published or there are future secondary uses of the data), then it will be necessary to obtain signed consent from each participant. Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script (see Appendix 1).

4. No incentives will be offered to the participants. *The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle. People volunteering to participate may be compensated financially e.g. for reasonable travel expenses. Payments made to individuals must not be so large as to induce individuals to risk harm beyond that which they would usually undertake.*

5. No information about the evaluation or materials is intentionally withheld from the participants. *Withholding information from participants or misleading them is unacceptable without justifiable reasons for doing so. Any projects requiring deception (for example, only telling participants of the true purpose of the study afterwards so as not to influence their behaviour) are deemed high risk and require approval from the relevant C-REC.*





6. No participant is under the age of 18.
Any studies involving children or young people are deemed to be high risk and require ethical approval through the relevant C-REC.
7. No participant has a disability or impairment that may have limited their understanding or communication or capacity to consent.
Projects involving participants with disabilities are deemed to be high risk and require ethical approval from the relevant C-REC.
8. Neither I nor my supervisor are in a position of authority or influence over any of the participants.
A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any study.
9. All participants will be informed that they can withdraw at any time.
All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script (see Appendix 1).
10. All participants will be informed of my contact details, and the contact details of my supervisor.
All participants must be able to contact the investigator and/or the supervisor after the investigation. They should be given contact details for both student and supervisor as part of the debriefing.
11. The evaluation will be described in detail with all of the participants at the beginning of the session, and participants will be fully debriefed at the end of the session. All participants will be given the opportunity to ask questions at both the beginning and end of the session.
Participants must be provided with sufficient information prior to starting the session, and in the debriefing, to enable them to understand the nature of the investigation.
12. All the data collected from the participants will be stored securely, and in an anonymous form.
All participant data (hard-copy and soft-copy) should be stored securely (i.e. locked filing cabinets for hard copy, password protected computer for electronic data), and in an anonymised form.





Project title: GMRoom - A resource-management tool for Tabletop Roleplaying Games -
Developing an accessible, intuitive and interactive web and mobile application.

Student's Name: Taylor Jordan

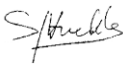
Student's Registration Number: 22201853

Student's Signature:  _____

Date: 03/11/2024

Supervisor's Name: Steve Huckle

I have read and authorised either (a) the participant information sheet and consent form or
(b) the Introduction and Debrief scripts. **Supervisor's Initials: SJH_____**


Supervisor's Signature: _____

Date: 4/11/2024 _____





A2. References

- ⁱ ‘50 Years of D&D: Why One of the World’s Most Iconic Games Still Matters - Axios Seattle’.
- ⁱⁱ ‘Study Finds Dungeons & Dragons Boosts Mental Health’.
- ⁱⁱⁱ ‘Character Classes for Dungeons & Dragons (D&D) Fifth Edition (5e) - D&D Beyond’.
- ^{iv} James, ‘Is It Easy to Play DnD?’
- ^v ‘D&D 5e’s Biggest Flaws as a TTRPG System – Artificial Twenty’.
- ^{vi} ‘RPG Campaign Planner | Customizing Your TTRPG Experience: How to Modify Game Rules and Create Homebrew Content’.
- ^{vii} ‘Tools and Features – D&D Beyond’.
- ^{viii} ‘Why Can’t We Create Homebrew Classes - General Discussion - D&D Beyond General - D&D Beyond Forums - D&D Beyond’.
- ^{ix} ‘Mod:Development - Roll20 Wiki’.
- ^x ‘The Best DM Tools of 2024: Finding the Tools for Your Table’.
- ^{xi} ‘What Is Full-Stack Project Management?’
- ^{xii} ‘Top Web Development Tools in 2025 | BrowserStack’.
- ^{xiii} ‘Home | Critical Role’.
- ^{xiv} ‘Baldur’s Gate 3’.
- ^{xv} ‘Waterfall Model: Pros and Cons (with Definition and Stages) | Indeed.Com UK’.
- ^{xvi} ‘Agile vs. Waterfall Project Management | Atlassian’.
- ^{xvii} Atlassian, ‘Kanban vs Scrum’.
- ^{xviii} Atlassian.
- ^{xix} ‘Introduction to the DOM - Web APIs | MDN’.
- ^{xx} ‘JavaScript Frameworks and Libraries - Learn Web Development | MDN’.
- ^{xxi} ‘Your First Component – React’.
- ^{xxii} ‘Writing Markup with JSX – React’.
- ^{xxiii} ‘Imperative vs Declarative Programming’.
- ^{xxiv} ‘Preserving and Resetting State – React’.
- ^{xxv} Rupert, ‘UI = f(States)’; ‘UI Is a Function of State’.
- ^{xxvi} ‘Keeping Components Pure – React’.
- ^{xxvii} ‘Node.Js — Run JavaScript Everywhere’.
- ^{xxviii} ‘Is Node.Js Bad? - DEV Community’.
- ^{xxix} ‘TypeScript vs. JavaScript: Your Go-to Guide | Toptal®’.
- ^{xxx} ‘MongoDB: The World’s Leading Modern Database | MongoDB’.
- ^{xxxi} Minjire, ‘Understanding SSR in Next.Js and Its Benefits’; ‘Next.Js by Vercel - The React Framework’.
- ^{xxxii} ‘User Interface Design in Video Games’.
- ^{xxxiii} ‘Sharing State Between Components – React’.
- ^{xxxiv} ‘WebSocket vs REST: Key Differences and Which to Use’.
- ^{xxxv} ‘WebSocket vs REST: Key Differences and Which to Use’.
- ^{xxxvi} ‘Undo’.
- ^{xxxvii} ‘Recursive-Diff’.
- ^{xxxviii} ‘2823. Deep Object Filter - In-Depth Explanation’.
- ^{xxxix} ‘useContext – React’.
- ^{xl} ‘useReducer – React’.
- ^{xli} ‘Baldur’s Gate 3’.
- ^{xlii} ‘Worker Threads | Node.Js V23.11.0 Documentation’.
- ^{xliii} ‘Elixir (Programming Language)’; ‘Actor Model’; ‘Erlang (Programming Language)’.
- ^{xliv} ‘Erlang -- What Is Erlang’; Valim, ‘Real Time Communication at Scale with Elixir at Discord’; Team, ‘WhatsApp Using Open Source Erlang’.





^{xlv} Obregon, 'Journey to Fearless Concurrency — Rust vs. Go'.

^{xlvi} 'Cassandra Vs MongoDB Comparison | MongoDB | MongoDB'.

