Contributions to the Bookkeeping web application and proposal for a new data table component

Joseph Torsney
joetorsney@gmail.com

Abstract. This report outlines contributions made as a CERN Summer Student to the Bookkeeping web application on the ALICE O2 computing project. The technical details of the application and contributions are summarised and requirements are proposed for a new data table component within Bookkeeping. The data table is a core aspect of the application, and the requirements are informed and justified by research, user feedback, and developer experience.

1 Introduction

1.1 ALICE and O2 Computing

A Large Ion Collider Experiment (ALICE) is one of several detector experiments at the Large Hadron Collider (LHC) at CERN. ALICE investigates the conditions of the early universe, close to the big bang, where matter was in a state known as quark-gluon plasma. This state of matter can be reproduced in high energy collisions between heavy ions, such as Lead.

The ‘O2’ in O2 Computing stands for Online-Offline. It is the project responsible for most computing tasks directly related to the ALICE experiment, including Data Acquisition and Trigger (DAQ), Experiment Control, Quality Control, amongst others.

1.2 Bookkeeping

Bookkeeping is a web-app that interfaces with the rest of the O2 system. It allows the Run Control team to keep track of the configurations of the ALICE experiment, data taking, and the status of particle beams in the LHC, known as LHC Fills. Bookkeeping also enables communication between staff in the form of Logs, which in particular helps the handover between shifts.

1.1.1 Technical details

Bookkeeping is composed of a backend NodeJS server, which exposes HTTPS and gPRC Application Programming Interfaces (APIs), and a frontend Single Page Application (SPA) which communicates with the backend. As well as the SPA, the APIs are used by other clients within O2. The server uses ExpressJS [1] to expose the HTTPS API and uses the Sequelize [2] Object Relational Mapping (ORM) library to interface with the MySQL database and manage schemas and migrations. Both the front and backend use a framework custom to the O2 project. The frontend SPA runs in the browser using Javascript and a reactive framework based on MithrilJS [3]. The basis of building front end web applications in the framework is with components, which are Javascript objects that correspond to and are rendered into HTML. Using these frameworks allow the creation of complex websites with more functionality (hence 'web-application') than would otherwise be possible because components are easier to maintain and can be reused. Unit tests for Bookkeeping are run by MochaJS [4] and integration tests use Puppeteer [5], which tests front end behaviour by mimicking the user interacting with the browser.

2 Contributions

2.1 Software development lifecycle

Tasks on the O2 project are managed as tickets in jira [6]. This software aids in planning, prioritising and distributing tasks within the team. The project is hosted on Github and git is used as version control. Contributors submit pull requests (PRs) linked to tickets in Jira, which are reviewed and incorporated into the codebase by the code owners. On each PR, Github actions execute the test suites. This helps to ensure contributors meet code quality standards and to prevent software regression. I had to ensure that my pull requests were up to date with the main branch during the review process by rebasing and dealing with merge conflicts.

2.2 Log Tree Display

One suite of features I implemented was in the Log Tree Display. As previously mentioned, logs allow users to communicate with each other. When a user navigates to the page to display a log, they are presented with the Log Tree Display, which highlights the selected log and displays the rest of the conversation in the form of parent or child logs (figure 1).

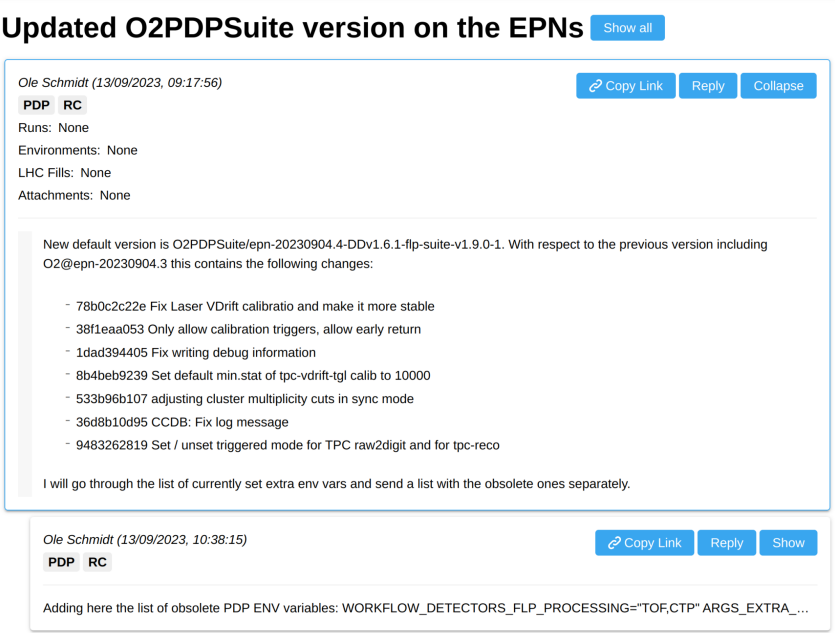

Figure 1: A screenshot of the log tree on the log details page in Bookkeeping. The log the user clicked on to get to the log details page is highlighted and expanded so its content is visible. A reply to this log is shown underneath. Note the copy link button available on each log.

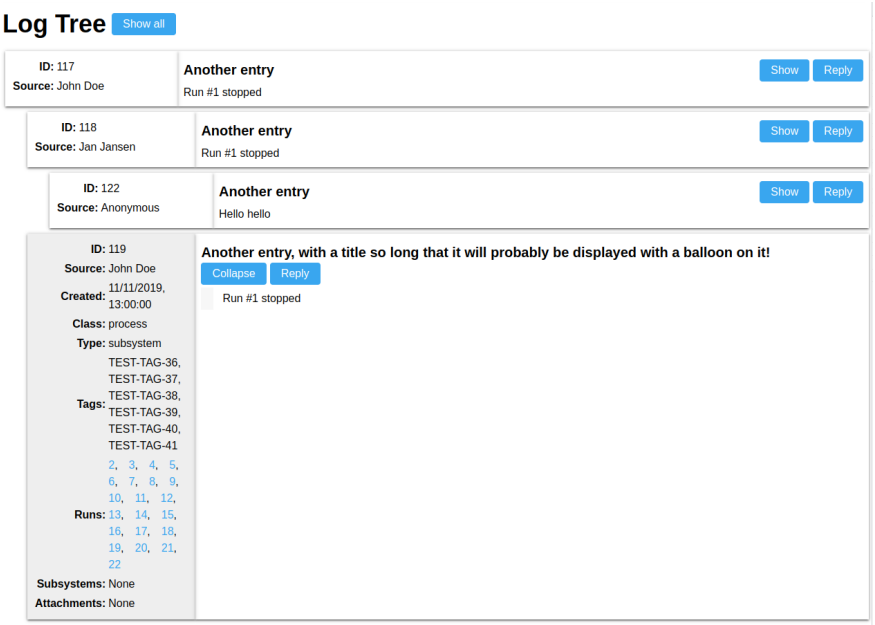

Figure 2: An example of the log details page in a development environment before I made changes to the log components. Log titles are duplicated, and expanded logs create a lot of wasted space due to the column of information to the left of the log body.

Over several iterations, I modified the layout of the cards representing logs to contain less wasted space, and hide duplicated information across cards. For instance, if child logs have the same title as the parent log, the title is hidden on each log component and displayed once at the top of the page. Figure 2 shows the log tree view before I made these changes, which can be compared with figure 1. Further improving the user experience, I also added a button to the log (figure 1) which copies the url of the log to the clipboard. The button is also able to notify the user when the copy was successful by temporarily changing its icon and text.

As part of these improvements to the UI, I also refactored the Log component in the codebase. Logs can be expanded or collapsed to show extra information such as tags or other associated entities. Both states were rendered using different components so the code, including the structure of the HTML and styling, was duplicated. This was problematic since the same changes to the code were made in two places. By factorising structure common to both expanded and collapsed log components into a parent component, the code was made simpler and more maintainable.

2.1 Pair Programming & Shadowing

In the initial stages of my placement, some tickets were completed with the help of the other summer student on the team. One such ticket was to add a column to the Runs Overview table (figure 3). I found this experience helpful in becoming more familiar with the project and the codebase, as they had completed a similar feature previously.

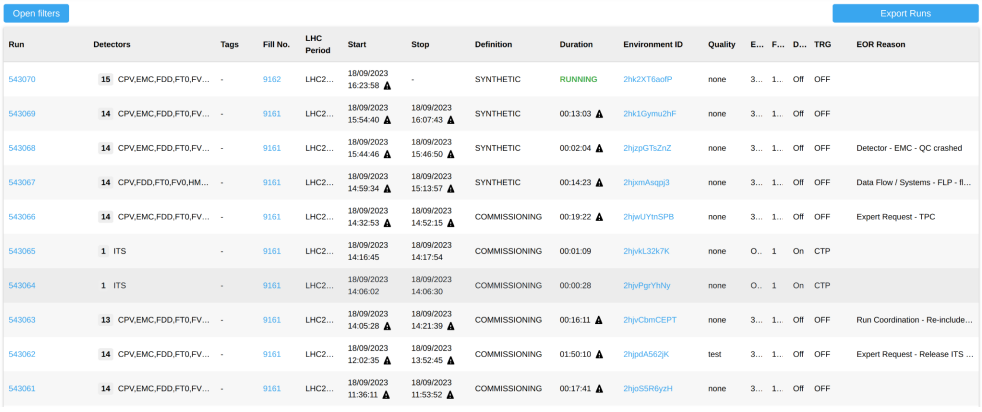

Figure 3: The Runs overview page in Bookkeeping displaying a list of Runs in the form of a data table. The final column ‘EOR Reason’ was added in a pair programming task.

I spent several days over the duration of my placement working from the Run Control room on ALICE at CERN Point 2. I had the opportunity to understand how Bookkeeping was used in operations at the control room, gather feedback from users about what could be improved about Bookkeeping, and watch how the ALICE experiment is managed. This was a valuable experience which gave me a context to consider when developing and researching new features.

2.4 Linking Logs to LHC Fills

A high priority feature in Bookkeeping was to be able to associate Logs with LHC Fills. This required both front and back end work and therefore had to be implemented incrementally over several git branches. This involved ensuring my branches were up to date with the main branch as I waited for reviews and worked on other features.

2.4.1 Modifying the database

Since Logs can be associated with many LHC Fills, and LHC Fills can be associated with many Logs, the relationship between them is 'many-to-many'. In order to represent such relationships in databases, a junction table is needed. In this case, the table *LogLhcFills* stores pairs of *log_id*'s and *fill_numbers*, the respective primary keys in the *Logs* and *LhcFills* table (figure 4).

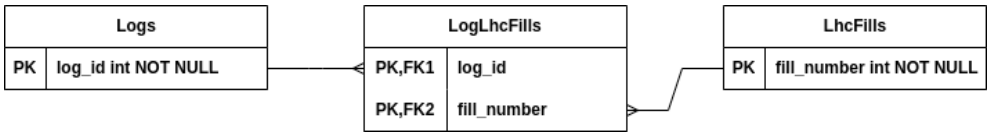

Figure 4: An entity relationship diagram to show the way in which the many-to-many relationship between Logs and LhcFills is implemented in the database. Keys other than the primary keys for Logs and LhcFills are omitted.

Sequelize works by allowing the developer to define models in Javascript corresponding to tables in the database. The Sequelize API also supports the several types of associations that a model can have (many-to-one, many-to-many, one-to-one), so that the SQL query can be generated appropriately. To define the relation I implemented a *LogLhcFill* model and specified the many-to-many association between Logs and LHC Fills. The primary key of this table is a composite of the foreign keys from the *Logs* and *LhcFills* tables *log_id* and *fill_number* respectively. Specifying the foreign key constraint is a way to ensure database integrity, meaning that when a new record is added to *LogLhcFills*, it is ensured the *log_id* and *fill_number* exist in their respective tables, otherwise the SQL query will fail.

2.4.2 API routes

In order for the backend to support the creation of logs with LHC fills, the appropriate API routes needed to be modified and backend functions created to propagate the request to the database. The flow of data through Bookkeeping after an API request is shown in Figure 5.

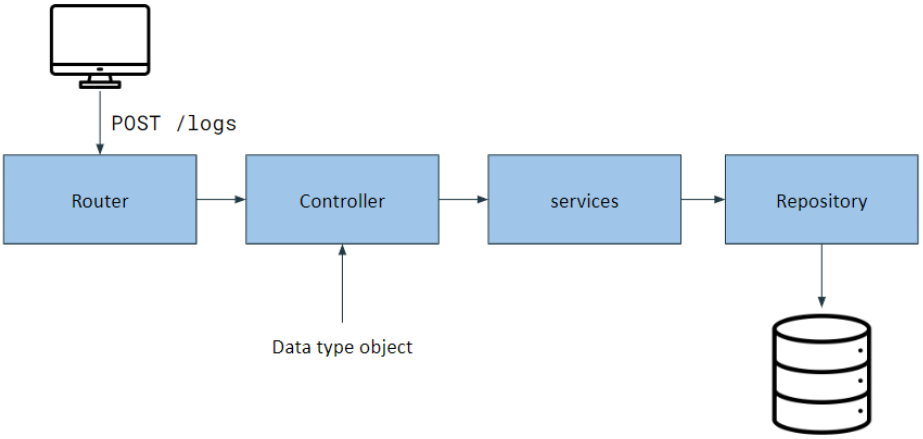

Figure 5: The flow of data through the Bookkeeping server application after an API request. The client sends a request to `/logs` with method `POST` to create a log.

In the Bookkeeping tech stack, API requests such as *POST /api/logs* are handled by routers, which pass the request to Controllers. Controllers do not perform any domain specific operations. They only extract the body and parameters from the request, verify that these are properly formatted against a specific data type object. In the case of creating logs with LHC Fills, the Logs Controller checks the LHC fills parameter is an array of numbers. Controllers pass this information to the service layer. In Bookkeeping, the service layer performs domain specific operations that do not interact with the data layer, or with the user input/output. To create logs with LHC Fills, I modified functions in the service layer to accept logs with LHC fills, checking that the LHC Fills specified in the request exist in the database. Services then query the database via Repositories which build and execute SQL queries with Sequelize and handle data persistence. At this step, I simply included the association between Logs and LHC Fills in the query to ensure that an entry would be created in the *LogLhcFill* table. To allow the user to filter by logs in the Log overview page, a similar set of additions were made to the *GET /api/logs* to control the logs that the route returns.

2.4.3 Frontend

The work on the back end performs most of the heavy lifting on this task, but without a way for the user to trigger an API request, it is useless. Bookkeeping uses a Model-View architecture on the frontend. Essentially, the model layer handles the application state and interacts with the server, whilst the view specifies only how to render the model layer, and notify the model when the user interacts with the view. The choice to use this architecture was enabled by MithrilJS as, unlike other reactive Javascript libraries, MithrilJS doesn't enforce any particular software architecture. An input field on the log creation page for entering LHC fill numbers was added, where the page model is notified of any changes. The model formats the user input by converting a comma separated string into a number array before configuring and making the API request. Similarly, I added an input to the

filters panel to enable filtering on the Log overview page, and finally a component on each log to display hyperlinks to the associated LHC fills.

3 Proposal for a data table component

3.1 User feedback

In Bookkeeping there is an Overview page for most of the entities, including Logs, Runs, LHC Fills, and Environments. These pages display a table listing instances of the entity and their properties, as well as a filters panel to be able to see instances that meet certain criteria. Therefore the overview pages, and in particular the table component, are core aspects of Bookkeeping from the users' perspective.

While shadowing at Point 2 during my placement, I had the opportunity to discuss and gather feedback from users. There were several points raised relating to the data table. When there are too many columns on the table for the width of the screen, the data in the table is clipped, meaning the user has to resize the window to make it visible. This would be alleviated, for example, by enabling horizontal scroll or allowing the user to hide or show the columns they desire. Another issue is due to the filters panel size. When there are many categories to filter the data by, the filters panel extends off the bottom of the screen, requiring users to scroll vertically. The width of the panel is also problematic since it is set to be a percentage of the browser width meaning on small screens the panel is too narrow. Finally, when the filters panel is open, displayed on top of the data table, hiding the data.

3.2 Developer experience and API

From a developers' point of view, creating and modifying instances of the current table component is a lengthy process, as I found when I added a filter to the Runs overview page. This is because there is no standard pattern to add filters and sorting. When creating a filter component in the filters panel, the developer must implement their own FilterModel class and ensure that all the necessary methods are implemented to manage the state of the filter. The developer must also write the URL parameters themselves, ensuring that column sorting is applied. The fact that most of the work is on the developer to implement filters and column sorting, instead of by extending base classes for example, has led to inconsistencies in the different overview pages from a users perspective. Some columns on the Runs overview page do allow sorting, whereas on the Logs overview page no columns allow sorting. Thus, harmonising the API for developers is desirable and would improve the user experience by reducing inconsistency.

3.3 Research

As well as user feedback, I conducted research into data table components from existing component libraries and examples on live websites.

3.3.1 React data table component

The React data table component [7] is an open source component for ReactJS with around 90000 weekly downloads on npm. Notable features include sorting and pagination out of the box. In its most basic use, the table takes two arrays: an array of data objects to display and an array of column objects (figure 5). Each column object has a name to be displayed as the column header, and a *selector* function, which takes a row of data and returns the data to be displayed in the column. This is to allow the developer to create columns that do not exist as properties on the data. For example, a column may need to display a success or fail badge based on several properties in the data. The popularity of this component demonstrates that its use of such an API is flexible and able to be integrated into many different web-apps in different business domains. Similar APIs are used in other popular open source components [8][9].

```
import DataTable from 'react-data-table-component';

const columns = [
    {
        name: 'Title',
        selector: row => row.title,
        sortable: true,
    },
    {
        name: 'Year',
        selector: row => row.year,
        sortable: true,
    },
];

const data = [
    {
        id: 1,
        title: 'Beetlejuice',
        year: '1988',
    },
    {
        id: 2,
        title: 'Ghostbusters',
        year: '1984',
    },
]

function MyComponent() {
    return (
        <DataTable
            columns={columns}
            data={data}
        />
    );
};
```

Figure 5: An example on instantiating a react-data-table component. Note how columns are defined independently from the data, and that each column has a selector function to compute column data.

3.3.2 Live Examples

Part of my research also involved looking at live examples on websites and apps, and how they deal with aspects such as including filtering and displaying large amounts of data. When browsing products on e-commerce sites, for example the outdoor brand Alpkit [10], I found that filters were often displayed to the side of the data instead of covering it (figure 6). In the example in figure 6, each category in the filters panel is in its own dropdown, hiding its content. Whether this should be implemented in Bookkeeping should be decided, as it reduces the amount of clutter on the screen, focusing the users attention on the data, but at the expense of an extra click each time the user wishes to modify a filter.

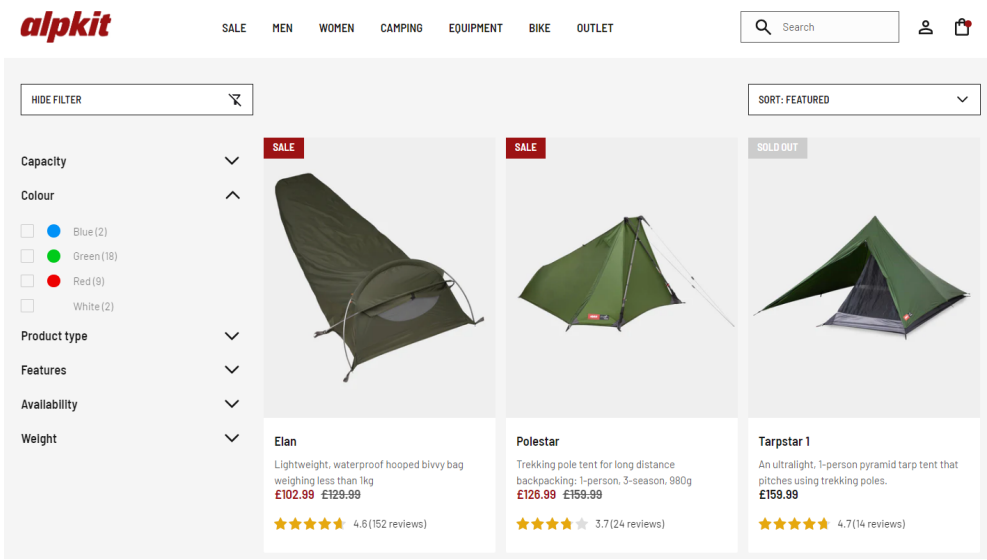

Figure 6: A product page from the outdoor brand Alpkit with the filters panel shown. Each category has a dropdown, and the entire panel is able to be hidden with the 'Hide Filter' button.

Although the data in the example in figure 6 is displayed as cards, as opposed to being tabular, the technique used where the filters panel is collapsible and not covering the data can still be used in Bookkeeping.

3.4 Requirements

With user feedback and research in mind, I propose the following requirements in the format of user stories

- As a user, I would like to be able to set the visibility of columns, so that I can hide columns that are irrelevant to me.
- As a user, I would like to be able to horizontally scroll the table, so that data in cells of the table are not hidden on small screens.
- As a user, I would like a collapsible filters panel to the left of the table, whose width is not dependent on the width of the screen, so I can more easily use the panel.

And for developers, the following developer requirements

- The structure of the data and the way in which the data is rendered should be separated
- The API should be column oriented, and each column backed by a column model
- Filters should be independent of columns, and each filter backed by a Filter model
- The data table component should be designed to be composed with other components, such as the filter panel, and wrapped by other layers.

4 Conclusion

This report explained contributions made to the Bookkeeping web application, the technologies I worked with, and the Software Engineering concepts I applied during my internship at CERN. User feedback and research was gathered to inform a proposal for a new data table component in Bookkeeping. I hope to further the work discussed by beginning implementation of the data table as an open source contributor.

References

1. ‘Express - Node.js web application framework’. Accessed: Sep. 22, 2023. [Online]. Available: https://expressjs.com/
2. ‘Sequelize’. Accessed: Sep. 22, 2023. [Online]. Available: https://sequelize.org/
3. ‘Mithril.js’. Accessed: Sep. 22, 2023. [Online]. Available: https://mithril.js.org/
4. ‘Mocha - the fun, simple, flexible JavaScript test framework’. Accessed: Sep. 22, 2023. [Online]. Available: https://mochajs.org/
5. ‘Puppeteer’. Accessed: Sep. 22, 2023. [Online]. Available: https://pptr.dev/
6. Atlassian, ‘Jira | Issue & Project Tracking Software’, Atlassian. Accessed: Sep. 22, 2023. [Online]. Available: https://www.atlassian.com/software/jira
7. React Data Table Documentation. Accessed: Sep. 22, 2023. [Online]. Available: https://react-data-table-component.netlify.app/?path=/story/getting-started-intro--page
8. O. Folkerd, ‘Tabulator - Interactive JavaScript Tables’. Accessed: Sep. 22, 2023. [Online]. Available: https://tabulator.info
9. ‘React Table component - Material UI’. Accessed: Sep. 22, 2023. [Online]. Available: https://mui.com/material-ui/react-table/