BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2017
http://excel.fit.vutbr.cz

Table Library for Data-intensive Applications

Gabriel Branderský*

Abstract
The default HTML table is rarely sufficient for data-intensive applications. Even feature-rich table libraries may not be designed for such applications. Therefore, this paper discusses the design of the table library with a different approach targeted for data-intensive applications.
The simplicity of HTML is the result of its declarative syntax. Since it is desired to make the usage of the library as easy as possible, the final table is available as a custom HTML element. This is where the Angular 2 framework comes into play; its components are not only usable through extended HTML syntax, but also have the means of communication among themselves.
The philosophy of the library is to allow developers easily disable or replace the parts they do not need. This is achieved by APIs with different levels of abstraction.
The library was successfully utilized in a customer-relationship management system. It is also published as open-source on Github for Angular 2 framework.

Keywords: Web Technologies — Single-page application — Front-end frameworks — Angular — Reusable components — Data table — Data-intensive applications

Supplementary Material: Source Code on Github — Demo App

*xbrand04@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

The motivation for this library emerged from the need of a real-world application started in the beta version of Angular 2. Existing libraries could not satisfy all project requirements. It was concluded that it should be supported by a library that is targeted for data-intensive applications. There are not many libraries that state this as their primary goal, but there are many ways how to handle the problem (depicted in the following paragraph).

Let us assume some relational database technology for storing data, which is a common practice in web development. Then the database schema may contain dozens of tables. Every table may have dozens of fields and thousands of records. In this data volume, all shapes of data could be found, figuratively speaking. So data complexity is another concern. The data must be presented in a way that is the most appropriate to the user. The library aims to handle high data volume and complex data by UI elements that reduce data overload on a user. Overall, the user gets more control over the displayed data.

One of the outstanding current solutions is ag-grid [1] where "ag" stands for agnostic. It is compatible with many different technologies, e.g. Angular, React, Web Components or pure JavaScript. The project strives to be the world's best JavaScript data grid for Enterprise. The entire ag-Grid company is devoted to this goal. Several license types are offered by the company. Nearly everything is included among the features of ag-grid, e.g. sorting on columns, filtering rows, selection of rows/cells, and grouping by values.

However, the ag-grid may be too heavy for smaller business applications. A lighter alternative is ngx-datatable [2], which is available for Angular 2 and beyond. It creates virtual DOM to handle large data sets. It also has intelligent resizing of columns and vertical and horizontal scrolling.

The last library ngx-datatable is very close to what we are aiming for in this paper, but with a different approach and varying features. For example, we try to avoid horizontal scrolling by allowing a user to dynamically change currently displayed columns. Another important difference is the philosophy. While

ngx-datatable optimizes for the most common case, we optimize for the edge cases. This can be seen in the handling of cells which commonly need to have customized look. So ngx-datatable offers a special directive for that, but no other elements except table cells can be augmented by that. On the other hand, we make each section of the table replaceable regardless whether it is a table header or cell. This makes the common case more tedious but enhances the overall flexibility.

The availability of different libraries is to the advantage of developers although the choice needs to be carefully considered. Since Angular 2 was released only recently (September 2016) and its ecosystem is not so rich yet.

Chapter 2 presents the UI of the table library without going into the details of the design process. We use the UI for reference in chapter 3, which gives an overview of the architecture. The iterations of the design process with their evaluation are discussed in chapter 4.

2. UI Design Driven by Requirements

We will start this section by listing the most important requirements. Then we present a sketch of the user interface fulfilling these requirements.

- Sort columns, either in ascending or descending direction.
- Change the order of columns.
- Display complex structured data inside one cell.
- Edit values in table cells, including complex values.
- Edit multiple columns, a.k.a. mass-editing.
- Allow user to set which columns are visible.
- Filter on table data.

Figure 1 shows a user interface sketch of the table library [3]. Firstly, visible columns are limited according to user preferences. Users can customize them dynamically in various ways. For enabling additional columns, there is a plus sign (marked with number 1) in the last header cell of the last empty column. Clicking on it would switch it to a select element (no. 2) with a list of available columns grouped into categories.

Additionally, each column has a drop-down menu (no. 3) with an option to add a column at the particular position next to it. In the sketch, the column is being added to the right from "Studies" column. For the reverse effect, there is also an option for removing a column. Two more options are for sorting columns in the desired direction. Column "Personal ID" is sorted from the lowest to the highest values. Since this is very common action, the column name can be clicked to sort or reverse the sorting direction.

Below each column name, a column-specific filter (no. 4) is available. Only the rows matching the filter conditions are shown. In our case, "Salutation" column restricts values to "Mrs." and "Doctor". Each column can have a different type of a filter. Besides selection filter, column "Name" can fuzzy search on a custom text.

Similarly, there are some specific dropdown options available only for columns with complex data types such as "Subfields" in the "Studies" column (no. 5). These options modify the display of column cells or more precisely sub-cells. Each study cell contains a list of studies for one person. One study is an object with multiple properties. By checking off a subfield, a corresponding property is displayed in the sub-cell.

Finally, the last dropdown option is intended for mass-editing of column values. The process is similar to editing a single table cell. Hovering over a table cell/sub-cell displays an editing icon that opens up a popover (no. 6) with an appropriate edit form. It contains either a single input field or as in the case of "Studies" several input fields for each study property. Mass-editing only requires an additional step, selecting of rows that should be edited (no. 7). The number of selected rows is shown in a drop-down option for mass-editing.

3. The Layered Architecture

The library adheres to the component-based architecture that is built into the Angular framework. A component is a basic building block in this architecture. Each component carries a specific responsibility and can be re-used by other components. An application is created by combining these basic building blocks into a component tree.

Diagram 2 shows the component tree for the table module. The root component `TableComponent` consists of two subcomponents, namely `TheaderComponent` and `TbodyComponent`. Each subcomponent is composed of even smaller subcomponents. The subcomponent `TheaderComponent` renders a table header while utilizing `AddColumnComponent` for adding new columns, and `ThComponent` to render a header cell. Similarly, `TBodyComponent` renders a table body while utilizing `TdComponent` to render body cells. Complex cell types are implemented in a separate component `ArrayCellComponent` or `ObjectCellComponent`.

At this point, the responsibilities for handling dif-

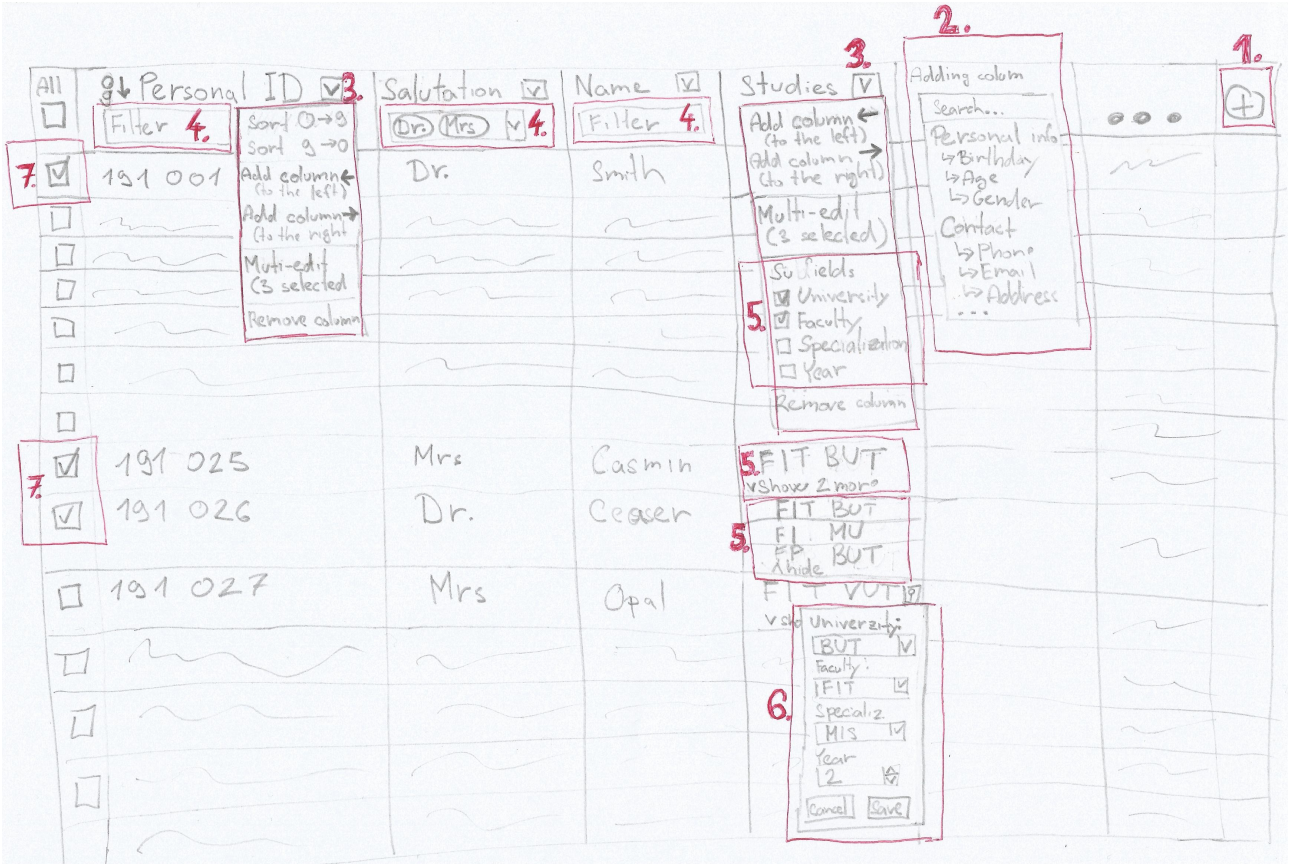

Figure 1. The UI sketch of the table library based on the requirements

ferent parts of a table are divided among the components. However, we did not define a public API that is accessible to the user of a library. Let us start by exposing only the root table component. It acts as a black box, while the other components are its internal details. The internal behavior of the table component is adjusted only through its inputs. In other words, the table component should provide an abstraction that is powerful enough to fulfill the requirements of the previous chapter for any variation of valid data. If we consider the inputs for sorting, the developer should be able to set a column that is sorted by default when a page loads. Additionally, it should be possible to disable the sorting on a specific column or disable sorting completely. It also makes sense to configure initial sorting direction when sorted by clicking on a label. In a similar fashion, we consider various inputs for other requirements (reordering, complex data, editing, column toggling and filtering).

What if a developer wants to slightly adjust the behavior of the table component, but there is no corresponding input for that? For example, there should be a column called “Actions” filled with buttons to execute some action on a row. Since this feature is application-specific, so it probably does not make sense to add it to a library. If not, the developer must replace the whole table, either by another library or by a custom implementation. This should not happen. It is very hard to create a good abstraction that anticipates all possible use cases. So what can we do about that? What if we create components with a different level of abstraction? That means we must carefully design inputs for all components from the table module and make them publicly accessible. This gives developers more flexibility for the price of learning additional application programmatic interfaces (APIs). The upper components have the highest abstraction level. To gain more flexibility, a developer can access building blocks of lower abstractions. He can then combine, complement or replace them as necessary. In that sense, the high abstraction API is just a shortcut that combines the lower-level components in a common way.

Apart from the subcomponents, the library offers utilities that help to join the individual parts together. Their presentation is outside the scope of this paper.

4. The Evaluation by Users & Developers

The user interface was created in several iterations. Each iteration ends with usability tests in order to find out what works and what does not [4]. In the testing session, users are asked to speak their mind while examining the interface. They also get a list of tasks to

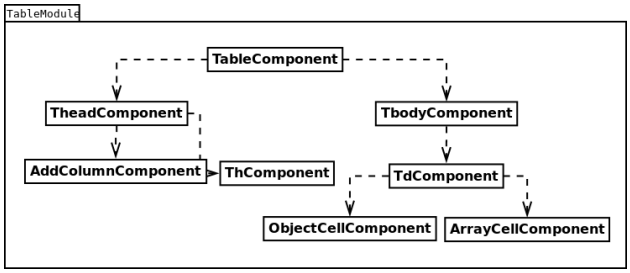

Figure 2. The partial diagram of the table library shows the decomposed architecture.

finish, e.g. find the youngest person. For this task, it is expected that the users discover the UI for sorting and how it works by themselves. This process can reveal even small usability problems. Finding problems has no value if they are not addressed by proper changes.

In the first phase, we start with a prototype [3], which lacks the underlying functionality (i.e. hard-coded outputs). Nevertheless, it verifies some of the basic assumptions made during the design without much development effort. All 4 users did not have any difficulty to accomplish their tasks.

In the second phase, the functionality is already in place. This time, the testing results in further polishing. Even small improvements can be of great value.

So far, we only mentioned the testing of UI, namely user experience. As we are concerned with the design of a library, we also need to care about good developer experience while working with the library. We employ the same testing methods, except that the users are developers who have to work with the programmatic interface in order to finish their tasks. Several important findings were made. The installation is considered overly complicated (because of dependencies). The common source of mistake was not passing output events upwards. This was probably expected because input parameters are automatically passed downward.

To conclude the evaluation, we would like to present the download statistics in Figure 3. The number of downloads is steadily increasing despite the missing propagation of the library (except for publishing it on the development platform " Github[1]" and the registry of JavaScript packages " NPM[2]").

5. Conclusions

In summary, we have created the table library starting with the sketch of the UI based on the requirements. Then we designed the architecture of the library. Finally, we evaluated both the UI and the architecture.

At the moment, the library has almost two thousand downloads altogether, and the number is steadily

[1] github.com

[2] https://www.npmjs.com/

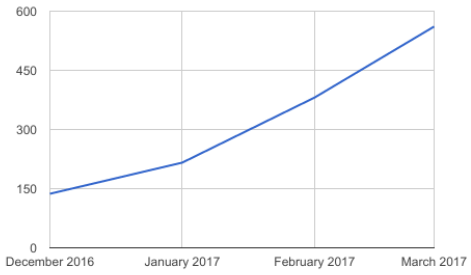

Figure 3. Download statistics for the library

increasing.

In the future, we plan to add various table extensions, e.g. (mass-editing) extension, exporting table data as CSV or PDF.

Acknowledgements

I would like to thank my supervisor Professor Adam Herout for the guidance, and constructive critiques.

My special thanks are extended to the members of the UX design agency "interfacewerk GmbH". They not only provided a real-world application of the library but contributed numerous ideas. Their support was truly indispensable.

References

[1] Javascript datagrid. https://www.ag-grid.com/. Accessed: 2017-03-23.

[2] ngx-datatable. https://github.com/swimlane/ngx-datatable. Accessed: 2017-03-23.

[3] Lukas Mathis. *Designed for Use*. The Pragmatic Programmers LLC., 2011. ISBN-13 978-1-93435-675-3.

[4] Steve Krug. *Don't Make Me Think: A Common Sense Approach to the Web (2Nd Edition)*. New Riders Publishing, Thousand Oaks, CA, USA, 2005.