TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

Daniil Kutsyna 144867IVSM

IMPROVING THE MAINTAINABILITY OF WORDPRESS PLUG-INS

Master's thesis

Supervisor: Juhan-Peep Ernits
PhD

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Daniil Kutsyna 144867IVSM

WORDPRESSI PISTIKPROGRAMMIDE HALLATAVUSE PARANDAMINE

magistritöö

Juhendaja: Juhan-Peep Ernits
PhD

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Daniil Kutsyna

08.05.2017

Abstract

Improving the maintainability of WordPress plug-ins

WordPress is a widely used website building platform that has a wide range of available plugins available in the marketplace from many different developers. On the other hand, WordPress marketplace today does not have a reputation as a high-quality software store. As the world wide web still grows rapidly, creating software that is maintainable will help to avoid a significant amount of user frustration and save developer time. Apparently, there are not many published studies containing in depth analysis of a JavaScript based software components and their architectures.

In the current thesis, we explore the power of different JavaScript analysis tools and analyse the design patterns of popular libraries and frameworks widely used in the context of WordPress. A case study is based on developing a piece of software in JavaScript twice using two different architectures. This enables us to compare the benefits of each development strategy and outline what worked and what needs another solution. Since the piece of software – a pre-flight planning tool, with calculations, which will be delivered in a form of WordPress plugin, it is supposed to have as little dependencies, as possible, to create the least amount of conflicts.

After development is finished the results will be evaluated, according to initial goals set, which concludes how effective this approach was and what was learned during the whole process.

This thesis is written in English and is 44 pages and 16 figures and 4 tables.

Annotatsioon

Wordpressi pistikprogrammide hallatavuse parandamine

Wordpress on laialdaselt kasutuses olev veebilehtede loomise platvorm, milles on wordpressi poe kaudu võimalik kasutada palju erinevaid pistikprogramme erinevatelt osapooltelt. Teisest küljest ei ole Wordpressi poel täna kõrge kvaliteediga tarkvara pakkuja mainet. Kuna veebilehtede hulk Internetis kasvab jätkuvalt, siis aitab paremini hallatava tarkvara arendamine vältida arvestatavat hulka kasutajate pahameelt ja säästa arendajate aega. Osutub, et ei ole avaldatud palju sügavuti analüüse JavaSciptis arendatud tarkvarakomponentide ja nende arhitektuuride kohta.

Käesolevas magistritöös uurime erinevate JavasScripti analüüsitööriistade võimalusi ja analüüsime populaarsetes raamistikes pruugitavaid disainimustreid, mida kasutatakse tihti ka Wordpressi kontekstis. Töös teeme juhuanalüüsi arendades üht JavaScriptis loodud pistikprogrammi kahest erinevast arhitektuurist lähtudes. See võimaldab uurida mõlema lähenemise eeliseid ja puudusi ja teha järeldusi, mis toimis hästi ja mis vajab veel paremat lahendust. Kuna loodav tarkvarakomponent – lennueelne planeerimistööriist arvutuste tegemiseks – luuakse Wordpressi pistikprogrammina, siis peaks sellel olema nii vähe sõltuvusi kui võimalik, et vältida konflikte teiste pistikprogrammidega.

Arenduse järgselt analüüsitakse tulemusi lähtuvalt algsetest eesmärkidest, mis võtab kokku pakutud lähenemise efektiivsuse ja mida me protsessi käigus õppisime.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 44-l leheküljel, 16 joonist, 4 tabelit.

List of abbreviations and terms

DPI	*Dots per inch*
TTU	Tallinn University of Technology
MI	Maintainability index
JS	JavaScript
WP	WordPress
MVC	Model View Controller
MVVM	Model View View-Model
MV*	Model View Whatever
DOM	Document Object Model
HTML	Hypertext Mark-up Language
CSS	Cascading Style Sheets
UI	User Interface
PHP	Hypertext Processor
API	Application Programming Interface

Table of contents

List of figures

List of tables

1 Introduction

This project involves the comparison of different data-flow patterns implemented in JavaScript and measured in the scope of it and then packaged as a WordPress plugin. This research evaluates the performance of different implementations and highlights the way software quality metrics reflect (or not reflect) certain architectural differences. Since the type of software is a plugin, which, to be commonly used have to be as independent, as possible (to exclude collisions in between dependencies versions, plus to be lightweight) is going to be developed with the pure JavaScript. And since most of the previous studies were done on a comparison of maintainability in between frameworks, this study will be focused on architectural choices and lack of pre-defined pattern, which most frameworks are bound to.

This section explains the motivation of current work and shows the light on research goals, at the same time current issues referenced.

1.1 Motivation

To understand better what could be done to develop a quality software proposed the development of a software piece, shipped in two different versions, written with the same programming language (and using the same standard version), accomplishing the same functionality.

Q1: How Data-flow patterns affect maintainability and predictability of the app?

Q2: What is the most suitable pattern for the plugin in JavaScript (JS)?

1.2 Goals

The primary aim is to create a software piece, which will fulfill all the needs of both customer and the end-user. At the same time software maintainability, code readability (software quality) is kept in mind, so that it will be possible to extend the product further. Another valid point and authors personal concern during development is to maintain the

product independent of different external libraries as much, as possible, since the software piece itself is supposed to be the plugin, which means that integration issues may occur. From the analysis point of view, interesting aspect is the reflection of the application architecture in different benchmarks, which could be used during development process from the very beginning (such as linters) and loading/rendering times, as well, as performance profilers, since plugin deals with live re renderings.

2 Related work

This chapter outlines current progress concerning the topic, concepts, and practices, along with most used libraries, pros, and cons.

2.1 Web application structure types

Front-end development and JavaScript (ECMAScript), in particular, evolved very rapidly during last few years[1][2]. With Single Page Application (SPA) concept introduced community with the help of certain companies and [3], I will call it "modular movement" concentrated on different parts of applications and applications started to consist of modules[4]. The current way of development web applications is by picking the modules[5][6], transpilers[7], task runners[8][9] and so on. Such techniques made development faster, but at the same time modules depend on each other so much, that sometimes happens something like this[10].

Let's assume that typical application structure is one of the most popular frameworks and libraries, but since there is no such database, so npm statistics by package[3] and github stars graph[2] will be used. The data highlights such packages like Angular and React. Even though jQuery is still the most used library throughout the web, it is not comparable in this case (Angular is a framework, with different layers, has a Shadow-DOM implementation and many other features). That mentioned, modern web apps consist of either Angular, which started from MVC framework and became MVVM or even MV* after version 2 and architecture changed by discontinuing usage of controllers and shift to component based design. Another option to be considered is React (which is just a library and stands for V - view in MV* model), but view library is not comparable to the framework.

The comparison scope here is an architecture of an application bound to data structures introduced by frameworks and libraries. In such scope React is supposed to have a Model part, which is simply a data layer library, such as Flux or Redux, plus some middleware to bind one to another, for example, react-redux[11].

2.2 Data flow patterns (data bindings)

Angular 1 (Angular JS), Angular 2 & Angular 4 use two-way data binding, which means that View layer can update the Model and at the same time Model can update the View. Such logic is easy to implement[12], as from the developer perspective, but it is easy to get lost when the application with such logic scales (Model updates one, or more than one View, this triggers another update and so on).

If there is only one Model and one View, application structure will look like:

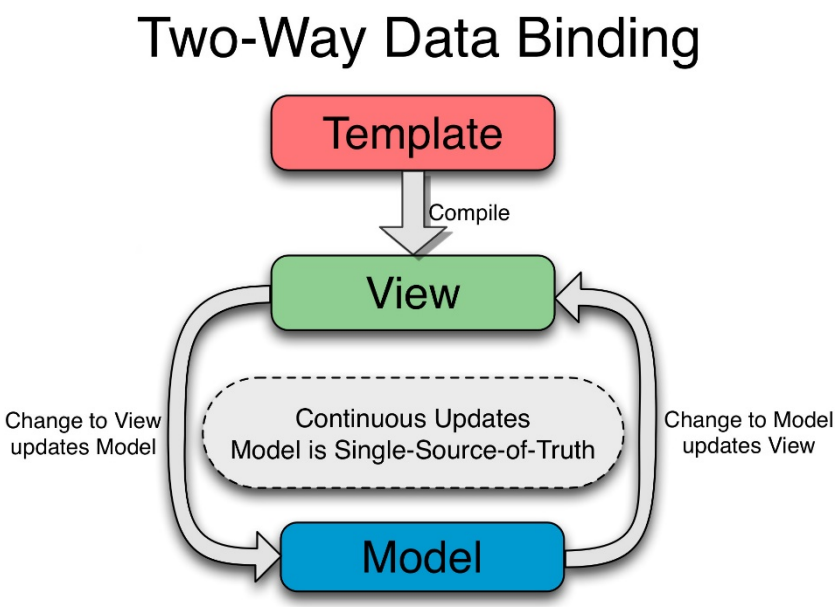

Figure 1. Two-way data binding angular graph[13].

To be able to compare React against Angular we need to add one of Data layer libraries, such as Redux of Flux to the bundle. It is also possible to use Redux with Angular, but such setup rarely used because of Redux is designed with the idea of unidirectional data flow behind it. Redux itself is the successor of Flux, so the idea of interaction with React is similar, but there are two main differences. First one is that Redux[14] uses single Store[15] and Flux[16] proposes using numerous of different Stores for the application. The second one is that Redux Store is immutable and after each action new object created, while Flux mutates the Store.

The Flux flow is given in [Fig.2]:

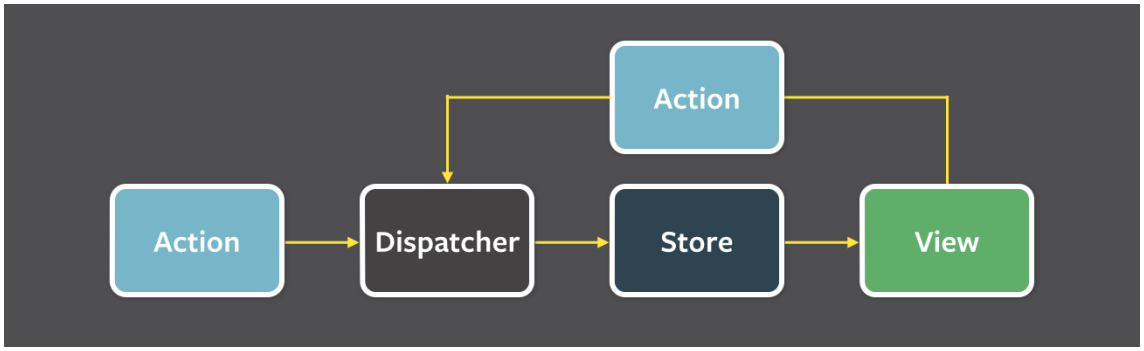

Figure 2. Flux Store[16].

Redux flow is given in [Fig.3]:

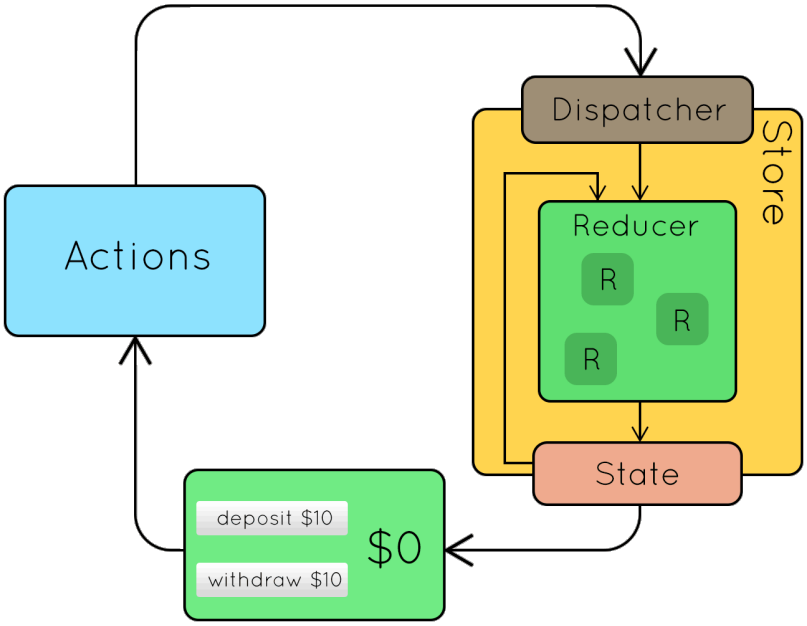

Figure 3. Redux simple data-flow[17].

Comparing Redux and Flux flow those are similar in terms of circular data-flow, except the number of Stores, Flux has multiple Stores, so the [Fig.2] shows one particular Event handling (because it is not possible to fire new events while firing another one). Redux Store also handles one change at a time (it could be full Store object swap, but it is handled by only one Reducer (or Reducers composition) at a time).

Since the concept of Reducers is supposed to deal with pure functions, it is not possible to do any server requests, for example. This part is supposed to be handled by Middleware and inside the State tree this event could be marked as ‘Loading’ state form

some variable and event listener could be assigned to dispatch a new event on receiving the response. This more complex case of the data flow shown in [Fig.4]:

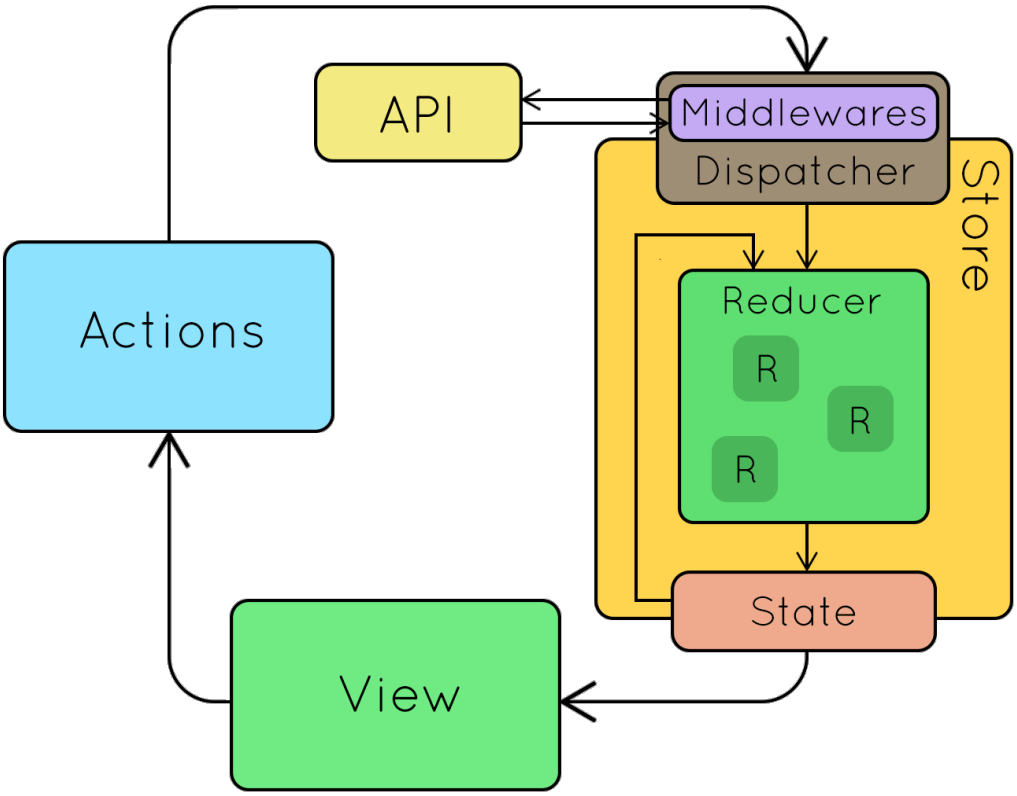

Figure 4. Redux complex data-flow[18].

3 Methodology

This chapter explains methodology and ideas behind the research, including analysis of previously introduced tools.

3.1 Previous Angular, React studies

As mentioned earlier Angular is MVC or MV* type of framework (depending on the version) and compared with React which should have Data layer and some middleware becomes MV*.

Analysis of those libraries and frameworks made with a focus on maintainability and scalability. Few authors used to discover maintainability concern[19][20] of Angular and React in particular before, by doing the analysis of TodoMVC (Todo lists made as a web-app in a variety of different libraries and frameworks, to sort of feel the taste of those) applications, which obviously are small ones and does not cover scalability issue, even though such apps show the basic idea behind each implementation tool (framework and/or library). During one of those studies, author transpiled TypeScript to JavaScript, just to be able to compare between two implementations, since Plato tool was not supporting TypeScript (which Angular team made preferable syntax, and it definitely has some influence, with such features as type inheritance, classes and much more). By doing it and not including any of benefits for maintainability of the codebase with TypeScript to quantitative research - questions the value of it.

Commonly used metrics for comparison of JavaScript frameworks are:

- Lines of code
- Cyclomatic complexity
- Halstead complexity

All three could be bundled into the one, called Maintainability Index (MI), let's review each of those in more details.

3.2 Lines of code (LOC)

LOC is a commonly used metric, but there are two slightly different types of it – Physical Lines of Code (also referenced as SLOC, an acronym of Source Lines of Code) and there is also LLOC (Logical Lines of Code). In this study SLOC will be used, it represents actual lines of code excluding comments. LLOC at the same time proposes to count only executable lines of code, also excluding comments[21]. The reason behind using this particular metric is that JS based software could have different scopes and references though files and in such case certain functions would be excluded, as nonexecutable from one file, even through it possibly executed on top of another one. Though it is necessary to exclude all the comments from files and since there are different ways how to write the code, in terms of leaving the whole function in one line or for example, use one line to set the name of the function, next lines to write the body of it and leave the ending curly brace to the next line, all the code will be passed through the parser[22], which will make the code style the same in all files, that will be compared. Another reason why this metric is included here – is that different software pieces will be compared and it is crucial to represent the scale of each and LOC is the metric which used for such estimates in software economics.

3.3 Cyclomatic complexity

Another common metric, which represents a number of linearly independent paths through each function[23]. In the context of this study, a metric is used to represent the code quality.

3.4 Halstead complexity

The empirical benchmark of software quality estimates the number of errors in a particular implementation[24]. Based on the number of distinct operator, the number of distinct operands and the total number numbers of those. Also used as a quantitative representation of solutions quality in this study.

3.5 Maintainability Index (MI)

Complex benchmark based on previously introduced ones counted as show in [Fig.5]:

```
Maintainability Index = MAX(0,(171 – 5.2 * ln(Halstead Volume) – 0.23 *
(Cyclomatic Complexity) – 16.2 * ln(Lines of Code))*100 / 171)
```

Figure 5. Maintainability index formula[25].

Proposed to calculate how easy is it to support and change the codebase, also used inside of MS Visual Studio and was used in the number of papers to compare JS based software, which is also the case in this study.

Even though those papers[19][20] came to a conclusion that Angular and React are almost the same regarding maintainability but mentioned that it very much depends on particular implementation. Methodologies [21][23][24][25][26] which authors were using don't include framework/library complexity estimation (those probably hard to quantify), and Data flow patterns not highlighted as a valuable asset in those maintainability estimations.

Addition metric, which was used in [27] paper is Page loading / Page rendering time. This is the one used in practice, quantitative[28] and considered during the page ranking, plus users notice the lack of performance capabilities, in case there are any.

3.6 Maintainability index of similar plugins

The similarity of the software, in this case, concluded by such criteria's:

- platform (WordPress in this case)
- category (table manipulations type of plugin)
- open source (to be able to compare the code with different tools, simply because minified code is not readable anymore and it affects MI, which is one of the core elements in this study).

3.7 Angular and React development patterns analysis

When AngularJS application grows model that was mentioned before[Fig.1] becomes such:

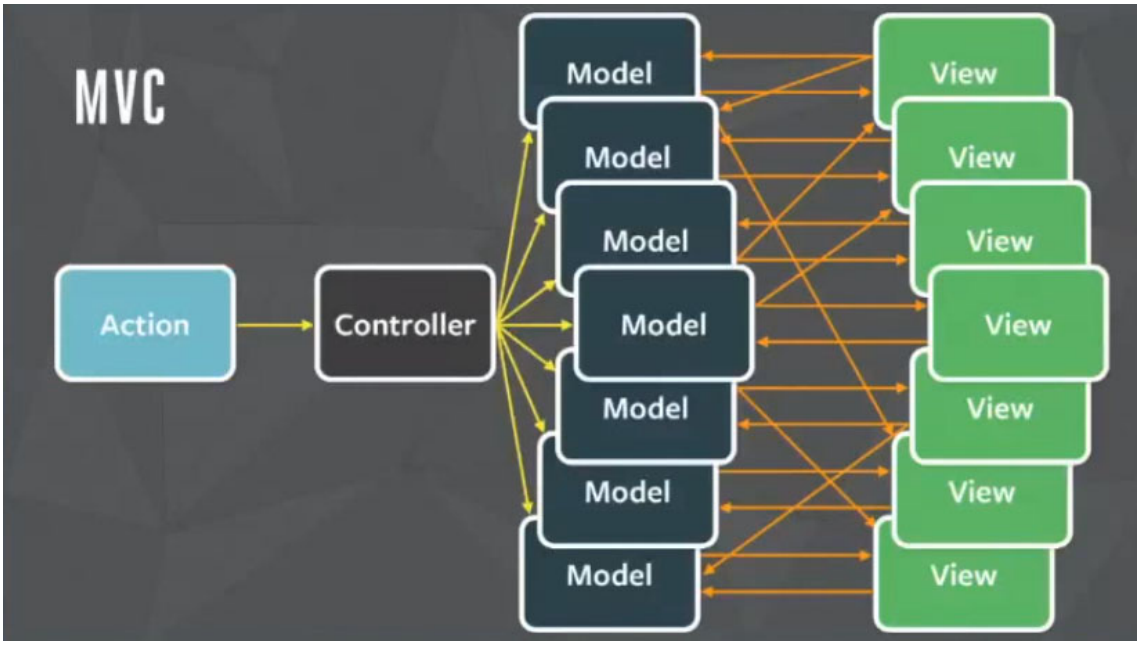

Figure 6. MVC JS application at scale[29].

It used to be during MVC pattern times, but later on, Angular team rewrote the product from scratch and implemented the View Model.

It is a separate layer in MVVM pattern, and by introducing it, Angular2+ becomes comparable to React since both have their implementation of it, e.g. Shadow DOM and Virtual DOM respectively. Both implementations made to optimize the performance of re-rendering View layer part, but implementations differ since the data flow is different.

At scale Angular2 application starts having multiple Views and Models, the structure becomes as shown on [Fig.7]:

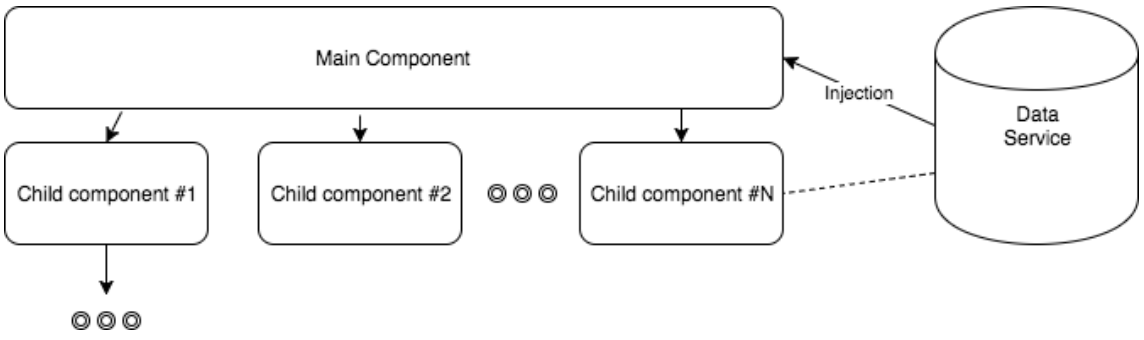

Figure 7. Angular MV* with many components nested.

React-redux structure for the same case is shown in [Fig.8]:

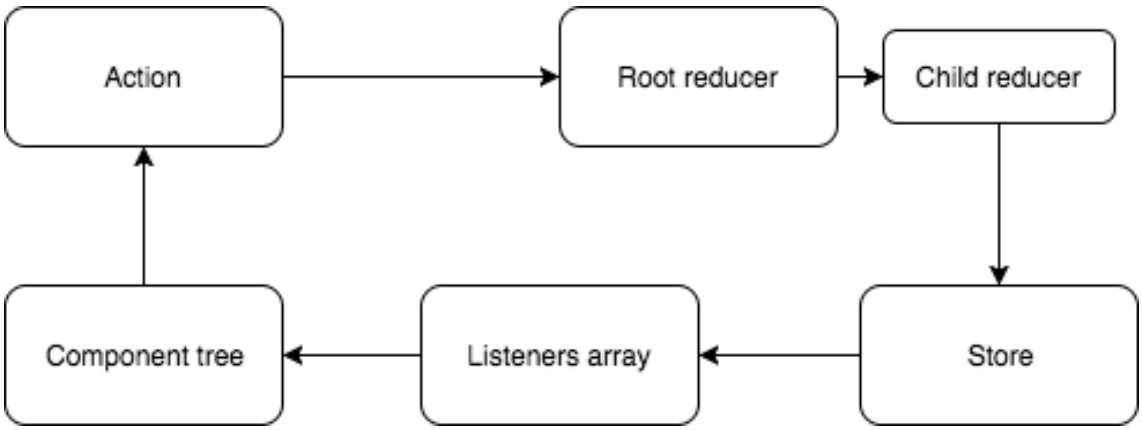

Figure 8. React data layer with many components.

While React Flux would look similar to Flux Store, which was previously mentioned and since Flux uses multiple Stores - it sums up with multiple micro apps inside.

3.8 Data layer analysis

Angular itself does not have a "single source of truth" kind of storage, the basic idea is that there are components with variables and each layer could have storage, or variables could be exported and imported in between components. One possible approach here is to implement a Store object tree at the top level component and inherit it from file to file, but it is obviously not going to scale. Angular is possible to integrate with Redux, but as was mentioned before - it is not a common practice. Moreover, the integration path with React is better due to UI treatment as a function of the state, based on which Redux can dispatch updates, based on actions[11].

Similar to data layer default representation, Angular with two-way data binding becomes very complex when scales, the event streams start to update the View Model and View, which is much better than it used to be with MVC [Fig.6], but it is still hard to debug. Understand which file used to dispatch that kind of action.

Redux, on the other hand, has the concept of reducers, yet at a scale, will probably end up with a number of components and many reducers for different types of events [30], but the reducers are pure functions and state is a single source of truth. It is easy to log at any point, load and save. Those possibilities definitely improve maintainability.

Debugging tools mentioned many times, as a reason why choose Redux, probably this is something could be added to maintainability scoring.

3.9 View layer

Component-based design became a common pattern in both React and Angular (starting from Angular2). Components are modules that could encapsulate each other. React contributor, and Redux co-author [31] to use two types of components: Containers and Presentational. Presentational components not supposed to include logic and just be a pure view. While Containers could have application logic inside, data behaviour and handle actions and callbacks for Presentational components. Such structure makes possible teamwork with people who will tweak style sheets of the components, and safer regarding nothing will affect the logic when the Presentational part changes.

Angular also supports similar approach, by separating complex logic to Services [32] (from requests to data exchange between components, usually asynchronous[33]). The problem there is the possibility to subscribe to changes on the other node, which at scale adds complexity to understanding and debugging of the app. Plus there is two-way data binding inside the Components, and at the end, Data flow becomes unpredictable unless each level is documented, which is rarely the case. However, even with documentation present, there is no guarantee the whole flow will be understandable and that somebody will be able to compile the whole picture.

3.10 Syntactic sugar

This aspect not considered, due to the initial requirement of resulting application being done based on most supported JavaScript standard, which is ES5 at this point [34][35].

The author would not reinvent the syntax of it, so this layer not highlighted in the study. The focus itself lies down on architectural choices.

Though coding styles will be considered as an issue and all the codebase will run through the pipe of few tools, which will be reviewed in detail later in this text. The idea behind those metamorphoses is to make all the codebase comparable, which means that it is necessary to restructure it by applying a set of rules.

3.11 Use case / Conclusion in current app development context

From the data flow patterns, at scale unidirectional data flow is either recommended by a framework[36] or by comparing the difference between models [29][Fig.6] to the one from which it started, like simple MVC data flow.

Implementation logic of one-way data bindings seems to be the best at Redux (the most deterministic one), comparable to the others (Flux and RxJS[37]) since Flux has multiple stores. Which should have a clear scope and use case at each level, at this stage the downside of Redux will be the traversal each time (it could be optimized with memorization techniques). RxJS is an implementation of FRP, and it has multiple stores, and from an architecture point of view, it has too much of asynchronous event handlers, and those are not the case in the current application. Furthermore, RxJS with async calls in mind makes the application less predictable. Redux also wins at the testing scope, since it has a possibility of loading the whole app state at any point, if render method is implemented in a certain way (downside is the limitations on render method architecture), which gives an opportunity to mock each interaction case easily.

From the event handling perspective, there are few parts - event listener bindings and reactions to those events (Observer pattern mentioned in Design Patterns[38] legendary book). Listeners could be bound to either each element (both, DOM and Data layer) we want to get changes of. In a case of DOM possible to use "event bubbling" - one of parent elements since the "bubble" will go up (by default) to the top of the DOM tree elements. In this case, all the DOM events are sent to the Dispatcher (Redux way), and then Action is Dispatched. This way changes would not be so chaotic. Furthermore, as suggested by JS Design Patterns book[39] in Vanilla a common practice is to include the Render method along Data, so when the data changes it will be passed right away to the Render method, which knows exactly how to handle it. Such pattern also involves less unnecessary traversals and easy understanding, since it is a common practice. To avoid duplications of such function there are few common ways, either having an array of pointers to DOM elements filtered by some criteria so that this array could be passed to the Render function after an update, the other way is to include the link to the Render method in each instance of such kind. Link trick is possible to implement by using simple Object assignment sign, which will not create a new one, but rather will include the link to the original one.

From the Data layer changes, it is a common practice to include Rendering method to the Data structure (in the case of pure JS)[39] and then functions evaluate when certain binding changes. It just gets the new data along with change event and re-renders the part with DOM binding.

4 Implementation

This chapter explains implementation in detail, including the development choices, with the reasons behind it.

4.1 Initial data inputs, requirements and limitations

The deliverable format of the plugin is a zip archive[41], folder structure best practices[42] outline similar structure to typical JS application and using the same principles of separating the files (modular), grouping by similarity and having folders for different types of files.

The plugin consists of HTML mark-ups, CSS stylings, JS files with most business logic implemented and PHP for WordPress integration and data saving.

Two versions of code will be implemented. First one will be delivered as is, by developing plugin in a straightforward way, without using any schemas before, based on understanding what is supposed to be done. This implementation will be useful during one of the last stages of this work – Analysis. The main purpose is to understand whether the measurement tools highlight certain issues, or those will pass silently and nobody will notice. The data flow of this version could be described as an infinite number of Observers and such approach definitely would not scale. Other than that, rendering handlers use similar approach in both versions.

Second version will be the version containing the patterns outlined before and will be described once more in the chapter below. The purpose of development in such order – is to keep away as much, as possible from reusing the same pattern twice and since this one seems to be the most thoughtful and organised one, straightforward approach will be done first.

4.2 Chosen pattern

Application architecture comparison ended up with choosing Redux-like store as a Model (in MV* terms). View layer is getting the updates from the Model by checking the

differences of the tree leaf's and applying render methods, for those with changes (Observer pattern) and passing a new value of the element, as shown in [Fig.9].

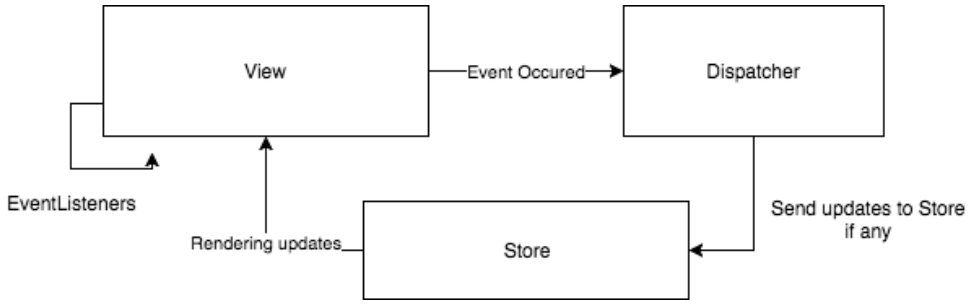

Figure 9. Application data flow schema of developed application.

With this architecture, it is possible to track all the changes that were done on the front-end and extend the application even further. Testing also becomes easier, since it is possible to recreate each interaction scenario and reproduce each state in particular.

4.3 Pattern with current data aligned

Pattern proposed by Redux fits well in the application developed, the only concern there is a lack of optimal renderer, which possibly will be the slowest part of this architecture. Except that proposed structure is very clean, compared to the 'ugly' version of the plugin, which is simply Observers everywhere and many DOM manipulations are triggered after certain changes were made (it's also the case in 'designed' version, but instead of DOM Observers, listeners were moved to the Store object, whenever possible). The final performance measurements will be presented in the Analysis chapter.

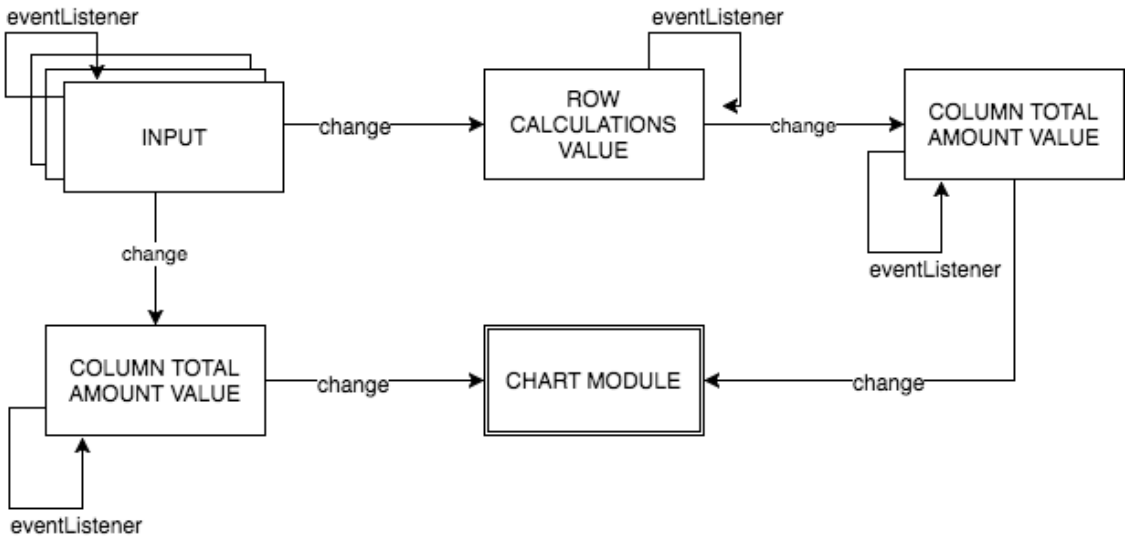

Figure 10. Data flow schema of 'ugly' version of the app, using Observer pattern.

4.4 Data layer

Store instance was implemented similarly to Redux Store concept. The Store is read-only, all the updates go through the Dispatcher, and all unknown actions result in no changes to the Store object.

To create the Reducer – 'Reducer Composition' concept was used, which stands for the main function, similar to a router, which then passes actions to the other, no combined Reducers.

```
var main = function(state, action){
…
switch (action.type) {
             case 'UPDATE_CHART':
                    return Object.assign({}, state, updateChart(state,
action) );
…
};
```

Figure 11. Reducer composition partial code.

Object.assign function and it's polyfills were used to achieve immutability of the state, in case of objects manipulations and Array.map(), along with Array.concat(), and Array.slice() methods were used to keep arrays immutable (Array.splice() for instance changes the original array).

Object.assign with the first parameter of an empty object implies that the returned object would not mutate the previous one. With arrays, not the most obvious way was used, but it is the functional approach, example of row removal from an array of the state:

```
var removeRow = function(state, action){
       return { tbody: state.tbody.slice(0, action.index)
                           .concat(state.tbody.slice(action.index + 1)) };
}
```

Figure 12. Reducer function example.

Implementation of those concepts guarantees that Reducer function is the only place for updates and with all the listeners subscribed to the Store it won't cause the unnecessary (or unpredictable) event occurrences. Which implies better readability and maintainability (at a list from the logical point of view, numbers will be at the Analysis chapter).

4.5 Event handlers

Dealing with events in JS there are few concepts that need to be considered. The two relevant concepts, in this case, are Observers pattern and Event Bubbling (and Capturing). Observer pattern is usually implemented by attaching event listeners to an element. For small applications, it is often possible to implement observed objects in separate and have only few listeners attached, while in bigger applications it is usually not the case.

Event Bubbling (or Capturing) is the way how triggered event propagates through the DOM tree. Let's assume we have a "click" event and some button inside of the body tag.

```
...<body><button type="submit" id='btn'>…</button></body>…
```

Figure 13. Binding to DOM element example.

After attaching the listener to the button in a usual way:

```
document.getElementById('btn').addEventListener('click',
clickHandlerFunction, optionalBoolean);
```

Figure 14. Event listener attachment example.

optionalBoolean will override in case of presence the default event Bubbling with explicit Bubbling or Capturing of the event.

clickHandlerFunction handles the event, it is also possible to stop event propagation inside of this function.

In case of Bubbling and click on the button click event will be first captured on the button itself and then will go up until the tree root, so that if there will be event listener on the body tag or any other button parent – all those elements will receive a click, unless stopPropagation method is used at the event inside of event handler function.

In the case of Capturing everything goes the same with stopPropagation, but event trigger will fall through the tree, so that if the button has any children elements inside, those will receive the click trigger.

This all simply means that in the case of nested clickable elements (in the case of handling the click events), it is necessary to be very careful when implementing the mechanism of interactions and keep in mind such JS specific concepts.

4.6 Rendering mechanism (DOM manipulations)

DOM manipulations are the slowest JS operations, which is known … so it is crucial to handle those in the least harmful way possible.

While working without diff-based algorithm and Virtual DOM of any kind, it was necessary to do as small (atomic) changes, as possible. It means that one rendering function for the whole Table, for instance, would always drop the focus (which affects the user experience) and also will need a lot of manipulations after each rendering. Such as attaching and detaching event listeners (if you wouldn't detach listeners it is possible to create a memory leak). To attach event listener – DOM traversal is needed, which is a rather expensive operation.

My approach here was to, first of all, used the Store as a main source of information (in 'designed' version of plugin) and then apply similar to the 'Reducer composition' concept to rendering methods, to group those by actions and filter only necessary ones (since after Store update, all the subscribed listeners get the new state and dispatched action objects).

The other way would be to call Rendering updates from the Reducers with the corresponding Action.type already, such pattern also mentioned in JS Patterns book[39], but it would ruin the purity of Reducer concept and would affect code readability, plus will affect the Data-flow scheme and app will become unpredictable after a certain amount of such changes.

4.7 Browser compatibility

Since pure JS was chosen to develop this software, the actual compatibility range is broad, application was tested myself in Safari 9, Firefox 53 (on Mac) and Firefox 49 (on Windows), Internet Explorer 11, Chrome for Android 48 and both Chrome 58 & 59, and Chrome Canary 60.

Most popular browsers were tested and both application versions were capable of doing what those were supposed to do, which means that those applications are ready for the further WP integration.

4.8 PHP integration

PHP is the language used inside the WordPress, to be able to integrate the developed JS application to WP it is necessary to chunk the files to pieces and to connect their appearance to the hooks. Hooks are such conditional statements that can be added to the code (like a subscriber for a certain event inside the CMS). There are two types of hooks – actions and filters. Action hooks are basically event subscriptions, for example, admin page is loading and then all subscribers are notified with this event. Filter, on the other hand, make it possible to manipulate the data, that WP provides a hook for. For example, when content of the page is rendered, filter could add share button to the end of it.

To add content to the admin menu – 'admin_menu' action hook is used, according to WP Codex[40]. Difference between non-admin version is that there is an editor for the table, e.g. buttons like 'Add row' and 'Save'. Users only get the table and chart with pre defined data and partially covered in inputs – table. And if data changes inside the table – chart get's updates.

Another function binded to action hook that was used – is wp_enqueue_style and the use case of it is obviously adding css files to template, if certain conditions are met (it is usually called from the inside of another action hook).

Similar practices are used inside on wp_enqueue_script, the difference comparing to previous one is just the fact that it is used to add scripts, not style sheets.

The way of adding the plugin content that I had in mind was a shortcode, because it is easy to use for the users. The user flow is that admin creates a table with some data and then shares it, or adds to the post or pages that he or she has in mind.

If the shortcode is using a template file, like in this case and getting it with the require(__FILE__) function, then the content initially will appear at the top of the post/page content. To prevent this from happening and display the content of the plugin deliverables in the place where shortcode is actually used it is necessary to use ob_start() method before require function and assign the result of ob_get_clean() to the variable after the require call. And then display that variable content, since it stores the actual template value.

It is also possible to pass the parameters while using shortcodes, in this case WP will be one of possible implementations is query the posts (if the content is supposed to be a post type of thing) with some parameters that are passed.

In case of passing the id, as a parameter, with let's say $post->ID, by adding the shortcode inside the webpage, as [shortcodeName id=42] and the we are querying the post type of 'aircraft', the handler function will look as shown in [Fig.15]:

```
function shortcode_name( $atts ) {
	global $wp_query, $post;
	$atts = shortcode_atts( array(
		'id' => ''
	    ), $atts );
	$loop = new WP_Query( array(
		'posts_per_page'    => 200,
		'post_type'         => 'aircraft',
		'orderby'           => 'ID',
		'order'             => 'ASC',
		'post__in'               => array( intval( sanitize_title(
$atts['id'] ), 10) ),
	    ) );
	    if( ! $loop->have_posts() ) {
	        return '<pre>Nothing was found</pre>';
	    }
	    $x = "";
	    while( $loop->have_posts() ) {
	        $loop->the_post();
	        $x .=
'<pre>'.get_the_title().'</pre><code>'.get_the_content().'</code><pre>'.$post
->code.'</pre>';
	    }
	    wp_reset_postdata();
	    return $x;
	}
```

Figure 15. Shortcode with parameter query example.

In this example either pre tag with "Nothing was found" will be rendered, or the same pre tag, but with post title, content and post id will be displayed. Usually there are more content to be displayed, but to make it shorter, this type of output was used.

To save the table versions the custom post types are proposed. The reason behind it – is that it is possible to get and ID for each of those posts and later on it is easy to query those and basically interact with those inside the admin area, with the usual to WP type of

interface. The shortcodes with ID parameter is easy to get from this type of binding, plus it is possible to extend this with custom fields bound once again to the post ID, so that different tables could use some more of custom parameters.

Other possible solution here is to create custom widgets. Those are better for themes which have many (or at least enough) of widget areas implemented, since the widgets could be placed only inside the widget areas. But this approach is less flexible, in terms of another extra dependency on the theme provider, even though there is a drag-and-drop interface of adding, deleting and moving those things inside of WP. The shortcodes could be also placed inside of widgets, which makes it a very universal type of thing. It is also possible to implement custom widget which will be the shortcode, with custom fields (or select input, instead of messing with ID numbers, it is possible to query all the ID's available and will make it even easier to use).

Saving and loading is another important part here, which is made by saving/loading the application state, to the content field, since it is very convenient, with the possibility to get and set (through POST or UPDATE) this field in JSON format and the target format of it is JS, which is easy to convert there and back. Communication will be done through AJAX requests, with content type of 'application/json'. Since the user won't be updating it manually (or at list I don't really see it as a convenient interface, because table and chart with buttons and inputs are easier to interact with, then finding the right field in JSON), it won't be displayed to the user, as opposed to usual post content.

So, the user is getting the custom post type with the custom editor view. The application (JS implementation) won't be modified a lot, since development with HTML, CSS and JS from the start, but would be split into pieces and loaded based on conditional WP hooks. Excerpts (short description of the post, similar to previews, limited by 55 words, by default) won't be shown at all, since the "content" is a JSON object of the application state and neither table, nor chart have a lot of text content that would be useful for such previews by any means. This type of post is not supposed to be seen, except while including the shortcode explicitly to the page or post content.

4.9 Final Packaging

Delivery format of the WP plugins is a .zip archive, in case the plugin is not yet in WP plugin catalog, a searchable marketplace (wordpress.org repository). To add the plugin to that repository it is necessary to register at wordpress.org and fill the form, which includes plugin name, description and an archive with the code. Then the submitted contents will be reviewed by WP moderators and a result notice will be sent back. On success, the SVN repository will be given.

For now, plugin will be delivered as an archive, but will be added to the marketplace later on, after reviewing all the code and user flow, for possible extensions, before the final release.

5 Analysis

This chapter concludes the previous work and quantifies all the assumptions in one way or another, answering to the initial research questions.

5.1 Maintainability comparison

As a comparison tool Plato[26] was chosen, it is basically a bundle of different tools (it includes such metrics like SLOC, Cyclomatic complexity, and Halstead complexity, plus counts MI based on previous ones), which were taken to evaluate JS code quality numerous times[19][20][27]. Plato itself based on complexity-report[43] and data obtained from two most common JS linters ESLint[44] and JSHint[45]. Complexity-report is wrapped around escomplex[46] ("Modular movement" which I've mentioned before).

Actual benchmarking tests were done with the suggestions Plato description included, filtering out just the source files (JS applications usually have a few settings files with .js extension, e.g. bundler setting or some other modules).

Filtering part was done by navigating to source folder and finding all the necessary files by executing bash command, as shown in [Fig. 16]:

```
find . –name *.js –exec mv {} . \;
cat *.js > __all.js

jsmin <__all.js >all.min.js

js-beautify -f all.min.js -o all.min.b.js

plato –d report all.min.b.js
```

Figure 16. Find and concat the code, beautify and count lines.

Then passing the data gathered to jsmin[47] to exclude all the comments and empty lines. I've used C file of jsmin and built the executable myself, by executing gcc jsmin.c –o jsmin. Then result of processing by jsmin passed to js-beautify[22], to exclude coding style differences and make code not "uglified" anymore, which affect the LOC amount. Then pass the file to plato tool and generate a report, with same coding style for all the files available and without empty lines.

The tool generated a report for each of those projects with many different criteria's overall and for each file in separate (MI, the number of lines of code, estimated errors in implementation (derived from Halstead complexity measures) and lint errors). Since the comparison is broad in a way, only MI and LOC were used in the first table.

For the first comparison, two other plugins were picked by (1) WordPress platform, (2) open-sourced code and (3) extensive JS manipulations with DOM. The goal of this comparison is to find out whether the software I created has a decent level of MI, to be considered as production ready[Table 1].

Table 1. Similar plugins comparison.

Plugin name	MI (out of 100)	SLOC
TinyMCE[48][49]	69.25	129544
ACF[50][51]	67.99	1842
Weight and Balance Calc.	66.05	593

5.2 MI comparison with different structured JS of plugins

Since the MI index is close to production software with 2m.+ and 1m.+ of users, two versions of the same software could be now compared in depth.

Table 2. Two developed version of plugin comparison.

Name	MI (out of 100)	SLOC	Avg. Load Time (out of 25) (ms)
Weight and Balance Calc. 'designed'	66.05	593	760.12
Weight and Balance Calc. 'ugly'	69.99	360	1016.08

Google Chrome Performance Profiler showed such performance benchmarks (average from 25 tries and min-max values, all the numbers are times in ms):

Table 3. Two developed versions of plugin average performance profiler comparison.

Name	Loading	Scripting	Rendering	Painting	Other	Idle	Total
Weight and Balance Calc. ‘designed’	20.49	443.85	34.85	5.87	100.57	137.62	760.12
Weight and Balance Calc. ‘ugly’	19.43	634.04	71.23	6.21	121.88	158.13	1016.08

Ending up with 760ms for designed version of average loading time including all the rendering, scripting. The fastest loading times were 624, 629 and 630ms and the slowest were 860, 861, 865ms, which leads to conclusion that those numbers are not accidental, since there are many close times during the test.

Second plugin version ended up with almost a 1s on average with the same amount of operations being done. The fastest times were 757, 849 and 873ms, which is way broader and it definitely has something to do with the way how the loading and rendering flow goes. The slowest were 1177, 1203 and 1307ms, which is also a rather huge difference.

That being mentioned, all versions were hosted at the same service, which is github and deployed as a gh-pages.

No bottleneck references were not found in the ‘designed’ implementation, except long loading of the google-charts library and long function execution, in particular. The second version of a plugin also has this issue, but it also has two more style recalculations with 102 elements affected appearing in case of 944ms loading at the point of about 620-640ms and another one at about 740-760ms. Which are highlighted as warnings and mentioned that there is a possibility of a bottleneck at those places. According to the graph above, scripting and rendering have the broadest difference. Since ‘ugly’ version has somewhat less LOC the loading part performs a little bit faster, but scripting part evaluates longer and rendering performs 2 times slower, most likely caused by those reflows.

Re-rendering performance comparison (done by applying changes to the table / usual workflow mock):

Table 4. Two developed versions of plugin average performance on updates.

Name	Add row	Increase input value by 1	Increase value by long press
Weight and Balance Calc. ‘designed’	30	76.33	74.85
Weight and Balance Calc. ‘ugly’	158	75.56	77.67

During those manipulations performance profiler was highlighting rendering operations as a possible bottleneck, since those are applied straight to the DOM.

Comparison by itself has mostly similar numbers, except the Add Row implementation, which is in second plugin performed way slower and handler function is highlighted in profiler, mentioning that this handler takes too long to execute.

Overall performance standing show that ‘designed’ version works better, loads faster and performs similar or better on rerendering, even though it has more LOC and by MI has a lower score by 3.95 points.

5.3 Outline

Numerous of metrics were referenced in this study and correlation between those is still unclear. MI represents static typing analysis side, but it has no guarantee whether the rendering or loading time will be optimal, those criteria’s become visible much later, when there is something to render and not while writing the skeleton code of the application.

On the other hand, all those tools (metrics and performance debuggers) were made to accomplish the same goal – enhance the code quality and overall performance of the end product.

Architectural choices are not measurable at the very beginning in terms of performance, not algorithms of course, but rather the data flow. Clean and easy to understand Observer pattern in case of straightforward implementation became hardly maintainable, because of a large number of cross-references between elements and such things must be

considered during development of applications with bigger scale then TodoMVC applications referenced in many previous works, while analyzing the standard patterns of popular JS frameworks. During this study, there were much more flexibility, which led to a decent amount of background research concerning each of the highlighted methods.

Redux architecture made reactive application structure clean and easy to understand, by applying a number of principles on top of Observables usage. For this study Redux-like architecture, in my opinion, was the right choice to do, according to the performance analysis, which was done on top of automated static code evaluation.

6 Summary

Data-flow patterns affect definitely affect maintainability predictability of the code, but it is not always represented by the numbers produced from available static code analysis tools. On the other hand, real-time performance metrics highlight possible sources of bugs and help during the late development stages.

The current study presented a long process of cross-platform development and numerous integrations, for this matter software is supposed to be designed thoughtfully and straightforward implementations will produce issues on different levels. To prevent this from happening all the tools mentioned above are worth noticing as helpers during development, to be able to produce a better-quality software.

From two implementations presented, the one with thoughtful architecture would scale better and faster, in case when there is no need to stick with just pure JS it is faster to create a valuable prototype with the help of libraries and then outline the architectural flows, since faster iterations would provide more of those benchmarking data sets (rendering/loading), which make more sense than error amount estimations and could highlight real bottlenecks of your application.

7 Future work

In the future, to improve the implementation it is necessary to upgrade the render methods with diffing algorithm or a Shadow DOM implementation of a certain kind. Since the bottlenecks, which were found are connected with rendering and it is definitely something that need to be improved in the future. Then the plugin will be easy to scale further, since the architecture itself is clean and there are many methods that could be reused, or dissembled and then reused to some extent (DRY pattern).

Further, it is possible to tweak around small architectural changes and measure the impact of those on rendering/re-rendering times, so that interactions will be delivered in the smoothest possible way and UX won't become an issue, but rather an opportunity to gain more users.

Chosen architecture also allows to include 'undo' and 'redo' functionality relatively easy, since the whole application state is represented as an object and all the rendering methods are in place, to add the possibility to tweak around the versions (or even versions filtered by action).

References

[1] Ecma Standards. 2017. Retrieved form http://www.ecma-international.org/publications/standards/Standard.htm

[2] Github stars history. 2017. Retrieved from: http://www.timqian.com/star-history/#facebook/react&vuejs/vue&jquery/jquery&angular/angular&emberjs/ember.js&angular/angular.js

[3] Download statistics for packages: angular, react, jquery, vue, ember. Retrieved May 09 2017 form: https://npm-stat.com/charts.html?package=angular&package=react&package=jquery&package=vue&package=ember&from=2013-01-01&to=2017-05-09

[4] Npm official website. 2017. Retrieved from: https://www.npmjs.com/

[5] create-react-app repository by facebookincubator. 2017. Retrieved from: https://github.com/facebookincubator/create-react-app/blob/master/package.json

[6] Angular quickstart reository. 2017. Retrieved from: https://github.com/angular/quickstart/blob/master/package.json

[7] BabelJS official website. 2017. Retrieved from: https://babeljs.io/

[8] Gulp official website. 2017. Retrieved from: http://gulpjs.com/

[9] Grunt official website. 2017. Retrieved from: https://gruntjs.com/

[10] kik, left-pad and npm blog post by Isaac Z. Schlueter. (March 23, 2016). Retrieved from: http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm

[11] Usage with React. Redux official docs. Retrieved May 10 2017 from: http://redux.js.org/docs/basics/UsageWithReact.html

[12] NgModel directive. Angular official docs. Retrieved at May 10 2017 from: https://angular.io/docs/ts/latest/api/forms/index/NgModel-directive.html

[13] Data binding. AngularJS Developer Guide. Retrieved May 10 2017 from: https://docs.angularjs.org/guide/databinding

[14] Three Principles. Redux official docs. Retrieved May 10 2017 from: http://redux.js.org/docs/introduction/ThreePrinciples.html

[15] Store. Redux official docs. Retrieved at May 10 2017 from: http://redux.js.org/docs/basics/Store.html

[16] flux concepts (Jan 19 2017) as a part of facebook/flux repository. Retrieved at May 10 2017 from: https://github.com/facebook/flux/tree/master/examples/flux-concepts

[17] Presentation “Redux from twitter hype to production” (slide 8) by Vyacheslav Pytel and Evgeniy Terpil. Frontend Dev Conf 2016. Retrieved 10 May 2017 from: http://slides.com/jenyaterpil/redux-from-twitter-hype-to-production/#/8

[18] Presentation “Redux from twitter hype to production” (slide 9) by Vyacheslav Pytel and Evgeniy Terpil. Frontend Dev Conf 2016. Retrieved 10 May 2017 from: http://slides.com/jenyaterpil/redux-from-twitter-hype-to-production#/9

[19] Magnusson, Erik, and David Grenmyr. "An Investigation of Data Flow Patterns Impact on Maintainability When Implementing Additional Functionality." (2016).

[20] Mousavi, Seyedamirhossein. "Maintainability Evaluation of Single Page Application Frameworks: Angular2 vs. React." (2017).

[21] Park, Robert E. Software size measurement: A framework for counting source statements. No. CMU/SEI/92-TR-20. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1992. (p.61 – p.71)

[22] js-beautify, JS Beautifier official repository. Retrieved 10 May 2017 from: https://github.com/beautify-web/js-beautify

[23] McCabe, Thomas J. "A complexity measure." IEEE Transactions on software Engineering 4 (1976): 308-320.

[24] Halstead, Maurice Howard. Elements of software science. Vol. 7. New York: Elsevier, 1977.

[25] Maintainability Index Range and Meaning (Nov 20 2007). Retrieved 10 May 2017 from: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/

[26] plato tool official repository. Retrieved 10 May 2017 from: https://github.com/es-analysis/plato

[27] Mariano, Carl Lawrence. "Benchmarking JavaScript Frameworks." (2017).

[28] Performance Analysis Reference by Kayce Basques. Retrieved 10 May 2017 from: https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/reference

[29] Hacker Way: Rethinking Web App Development at Facebook. (30.04.2014 F8 Conference, Facebook). Retrieved May 10 2017 from: https://facebook.github.io/flux/docs/in-depth-overview.html https://youtu.be/nYkdrAPrdcw?t=11m39s

[30] Performance. Redux official docs. Retrieved May 10 2017 from: http://redux.js.org/docs/faq/Performance.html - performance-scaling

[31] Presentational and Container Components by Dan Abramov (23 Mar 2015). Retrieved 10 May 2017 from: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

[32] Services. Angular official docs. Retrieved 10 May 2017 from: https://angular.io/docs/ts/latest/tutorial/toh-pt4.html

[33] Component Interaction. Angular official docs. Retrieved 10 May 2017 from: https://angular.io/docs/ts/latest/cookbook/component-communication.html

[34] ECMAScript 5 compatibility table. Retrieved at May 10 2017 from: http://kangax.github.io/compat-table/es5/

[35] Support tables for ECMAScript 5. Retrieved at May 10 2017 from: http://caniuse.com/#search=es5

[36] Building Large-Scale Apps. vue.js official docs. Retrieved 10 May 2017 from: https://v1.vuejs.org/guide/application.html

[37] RxJS Overview. Official RxJS docs. Retrieved at May 10 2017 from: http://reactivex.io/rxjs/manual/overview.html

[38] Gamma, Erich, et al. "Design patterns: Abstraction and reuse of object-oriented design." European Conference on Object-Oriented Programming. Springer Berlin Heidelberg, 1993.

[39] Osmani, Addy. Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide. " O'Reilly Media, Inc.", 2012.

[40] Administration Menus. WordPress Codex. Retrieved 10 May 2017 from: https://codex.wordpress.org/Administration_Menus

[41] Detailed Plugin Guide. Part of Plugin Handbook by WordPress.org (Jan 30 2017). Retrieved 10 May 2017 from: https://developer.wordpress.org/plugins/wordpress-org/detailed-plugin-guidelines/

[42] Best Practices. Part of Plugin Handbook by WordPress.org. Retrieved 10 May 2017 from: https://developer.wordpress.org/plugins/the-basics/best-practices/

[43] compleity-report tool official repository. Retrieved at May 10 2017 from: https://github.com/escomplex/complexity-report

[44] ESLint. Official website. Retrieved at May 10 2017 from: http://eslint.org/

[45] JSHint ver. 2.9.4. Online tool. Retrieved at May 10 2017 from: http://jshint.com/

[46] escomplex official repository. Retrieved at May 10 2017: https://github.com/philbooth/escomplex

[47] JSMin official repository. Retrieved at May 10 2017 from: https://github.com/douglascrockford/JSMin

[48] TinyMCE JavaScript Rich Text editor official repository. Retrieved at May 10 2017 from: https://github.com/tinymce/tinymce

[49] TinyMCE Advanced by Andrew Ozz. Plugin page on WordPress Plugins listing. Retrieved at May 10 2017 from: https://wordpress.org/plugins/tinymce-advanced/

[50] Advanced Custom Fields official repository for Advanced Custom Fields Wordpress plugin. Retrieved at May 10 2017 from: https://github.com/elliotcondon/acf

[51] Advanced Custom Fields by Elliot Condor. Plugin page on WordPress Plugins listing. Retrieved at May 10 2017 from: https://wordpress.org/plugins/advanced-custom-fields/

Appendix 1 – [Heading of Appendix]