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

Sergey Malyshev 153008IAPM

SIMPLIFYING A PROCESS OF BUILDING COMPONENT-BASED PROGRESSIVE WEB APPLICATIONS BY DEVELOPING A GUI TOOLCHAIN FOR VUE.JS MVVM FRAMEWORK

Master's thesis

Supervisor: Martin Verrev
MSE

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Sergei Malõšev 153008IAPM

KOMPONENTIDE PÕHISTE PROGRESSIIVSETE WEB RAKENDUSTE EHITAMISE PROTSESSI LIHTSUSTAMINE ARENDADES GUI TÖÖVAHEND VUE.JS MVVM RAAMISTIKU JAOKS

Magistritöö

Juhendaja: Martin Verrev
MSE

Tallinn 2018

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: Sergey Malyshev

07.05.2018

Abstract

Popularity of web development contributes to active adaption of new standards and technologies. One of those trending technologies is Progressive Web Apps (PWA) that look and behave like native applications but are platform independent and portable. MVVM GUI frameworks allow the creation of reusable components that can significantly enhance development of said apps.

The aim of this thesis is to describe a workflow and develop tools that can considerably simplify PWA development. The purpose is to optimize the component development workflow using Vue.js MVVM framework as regardless of its widespread adoption it lacks visual tooling.

During this thesis is developed a conceptual solution focusing on simplifying the existing PWA creation workflow using Vue.js framework. It covers all steps that are required to produce a complete PWA - component creation, component management, application building from these components and adding PWA specific parts. The thesis covers both theoretical background and concrete implementation. The theoretical part focuses on the process and methodology of PWA creation using the said framework. In the practical part the conceptual solution is implemented in the form of a prototype consisting of 2 parts – the component creator and the application builder.

This thesis is written in English and is 106 pages long, including 11 chapters, 17 figures and 24 tables.

Annotatsioon

Komponentide põhiste progressiivsete web rakenduste ehitamise protsessi lihtsustamine arendades GUI töövahend Vue.js MVVM raamistiku jaoks

Veebiarenduste populaarsus aitab kaasa uute standardite ja tehnoloogiate aktiivsele kohandamisele. Üks neist tuntud tehnoloogiatest on Progressiivsed veebirakendused (Progressive Web App - PWA), mis näevad välja ja käituvad nagu natiivsed rakendused, kuid on platvormil sõltumatud ja kaasaskantavad. MVVM GUI raamistik võimaldab korduvkasutatavate komponentide loomist mida saab palju aidata nimetatud rakenduste arendamisega.

Selle töö eesmärk on kirjeldada töövoogu ja töötada välja vahendid mida saab tunduvalt lihtsustada PWA arendamist. Eesmärgiks on komponentide põhiste arendamise töövoo optimeerimine Vue.js MVVM raamistiku kasutades, sest sõltumata tema laialdasest kasutamisest, temal puuduvad visuaalsed töövahendid.

Selle töö raames töötatakse välja kontseptuaalne lahendus, mis keskendub olemasoleva PWA loomise töövoo lihtsustamises, kasutades Vue.js raamistikku. Ta katab kõik sammud mida on vaja PWA tegemiseks - komponentide loomine, komponentide haldamine, rakenduste ehitamine kasutades komponendid ja PWA spetsiifilised osad. See töö hõlmab nii teoreetilisi tausta kui ka konkreetset realiseerimist. Teoreetiline osa keskendub PWA protsessile ja metoodikale. Praktilises osas rakendatakse kontseptuaalset lahendust prototüübi kujul, mis koosneb kahest osast - komponentide looja (component creator) ja rakenduse ehitaja (application builder).

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 106 leheküljel, 11 peatükki, 17 joonist, 24 tabelit.

List of abbreviations and terms

PWA	Progressive Web App: Regular web applications that look like native applications and have some features of them.
RAIL	Response, Animation, Idle, Load: A user-centric performance model that consists of the listed actions.
DOM	Document Object Model: API to manipulate a web page from JavaScript.
UX	User Experience: Person’s emotions and attitudes when he uses a software.
GUI	Graphical User Interface: An interface between a program and a user where the user interacts with the program using graphical elements like icons instead of writing text commands.
CSS	Cascading Style Sheets: A language used to describe a presentation of a web page.
HTML	Hypertext Markup Language: A language that is used to create web pages.
MVVM	Model-View-ViewModel: A design pattern that facilitates a separation of the GUI from the application logic.
JSX	JavaScript Syntax Extension: An language extension widely used in React framework.
OOP	Object Oriented Programming: A data-centric programming parading.
ES	ECMAScript: A programming language (JavaScript is one of its realizations).
UML	Unified Modelling Language: A general purposed modelling language widely used to visualize a design of software.
SPA	Single Page Application: A web application that uses only one web page to display all possible content instead of loading additional pages.
JSON	Java Script Object Notation: A text-based data exchange format that has syntax similar with JavaScript object definition.
URL	Universal Resource Locator: An unique identifier that is used to reference resources in the web.
CASE	Computer-Aided Software Engineering: A set of tools and methods using software to assist with engineering tasks.

API	Application Programming Interface: A set of rules or protocols to define interconnection between various software.
CRUD	Create, Read, Update, Delete: Acronym frequently used to define a set of the listed operations.
HTTP(S)	Hypertext Transfer Protocol (Secure): An application level protocol that is used to exchange data in the web. The secure version additionally uses SSL or TLS for traffic encryption.

Table of contents

List of figures

List of tables

1 Introduction

1.1 Locating the research area

When a developer starts his application development, he has to choose what platform to use. He chooses between a native application or an application targeted to a virtual platform. In the first case the developer will have full access to full functionality of the platform directly but the application will work only on the concrete platform. If the developer needs to use another one, then he must change all platform-dependent code. To help with this severe portability issue he can use cross-platform libraries. This typically results in additional code and size and still needs recompilation. In the second case the application is not targeted to the concrete real platform, but a virtual one and as a result can work without any changes on all platforms where this virtual platform exists. This solution greatly increases portability. But when the developer needs to deliver his application to an end user, the user needs to download it and also install the virtual platform to run it what complicates a deployment process a little bit. On the other hand, if you use the web technology, then the user does not have to install anything. Also web applications are very portable and do not require a user to install anything. To satisfy user needs browser developers actively add new features. As a result, web applications have become as powerful as comparable desktop applications in some aspects. One good example was Chrome OS [1]. Its approach is to run applications in the browser environment. This gave start to the new type of web applications. These applications can be used across multiple devices with multiple screens like desktop, tablets or mobile and may work offline and be installed as native ones. Such applications are called progressive web apps (PWA). Requirements to look and behave as an application but not as a web site requires changes to the GUI (Graphical User Interface) design process. Typical GUI of a web site consists of a number of block and inline elements. But these elements are too low level, while GUI of an application consists of a standard set of high-level elements (sometimes called widgets) such as menus, modal windows, toolbars, layout managers, cards. There is a clear need to extend standard web technologies with possibilities to build GUI using reusable components.

1.2 Existing solutions

There exist numerous frameworks and tools that assist with progressive web apps building. Service workers, icons and manifest generators help to transform your application into the progressive web application. There also exist platforms for website building like WordPress (https://wordpress.com) that support PWAs or special builders e.g. Appypie (https://www.appypie.com) targeted to PWAs construction. These tools are meant for general public and not for developers focusing primarily on presentation.

For developers there exist numerous JavaScript frameworks for component-based GUI development. Currently the most popular are React (https://reactjs.org), AngularJS (https://angularjs.org) and Vue.js (https://vuejs.org) [2]. React defines components, presentation and functionality in JavaScript. Running a React applications in a browser requires transpiling and unlike AngularJS and Vue.js it does not have out of the box data binding to power its MVVM (Model-View-ViewModel) approach. AngularJS is a MVVM framework with a two-way data binding, but it has own drawbacks like backward compatibility problems and complexity. It also uses TypeScript programming language what is an additional requirement and slows down the learning [3]. And there comes Vue.js. It is like a lightweight and simpler version of AngularJS. Vue.js can be placed to the 3-rd place after React and AngularJS. It is also the newest framework out of them. It was released in 2014 and it was called one of the most developing frameworks while AngularJS's popularity continues to fall. For example, AngularJS has about 60000 stars on GitHub, React has 90000 and Vue.js has more than 80000.

Previously described frameworks do not deal with appearance and responsive web design. Responsive means that the application has to display its content equally good on different screens. For these purposes there exist CSS (Cascading Style Sheets) frameworks and the most popular of them is Bootstrap (https://getbootstrap.com). It provides elegant responsive layout using grid system powered by the flexbox layout solution. Flexbox by itself is quite powerful, but Bootstrap's grid system deals with different screen sizes using break-points and provides a simpler and more clear solution using the rows and columns abstraction.

There also exist graphical developing tools for these frameworks. While React being the most mature and widely adopted has well supported tooling, Vue.js being the newest does not have such tools. There exists a GUI component building tool for React named Structor (https://github.com/ipselon/structor). It allows to create React components in a semi-graphical environment using Bootstrap CSS framework. Structor runs as a web application and is distributed via npm (package manager for node.js). Another example is React Studio that allows to build React applications in a semi-graphical mode, but it is accessible only on macOS.

1.3 Problem statement

As was shown before, there exist enough different frameworks for component-based web applications building. But as a result you have to deal with syntax and concepts of these frameworks and this requires additional learning and time. For example, you have to know native web technologies like HTML (Hypertext Markup Language) and CSS and in addition to them you have to know frameworks like Bootstrap and React. This means that this approach is complex and can be simplified. On the other hand, high level content management systems and site builders like WordPress do not provide enough control and an interactive view model for programmers.

There exist people who want to focus only on the core functionality of their application without dealing with the GUI part and get an interactive programming model of the application GUI with minimum efforts. Example of these people can be a student who wants to make an application for his non-trivial calculations and to use it on his computer or a phone. Another example is a developer who wants to make a GUI prototype for future development. These people are experienced in JavaScript, but not in the web GUI. Nowadays JavaScript is widely used outside the browser thanks to technologies like node.js [4]. But there still must be someone who deals with the GUI development and develops components using all functionality provided by browsers and frameworks and this part also can be simplified to assist these people. Maybe these people want to create some specific components only to use them somewhere else, but the result of their work can be shared to greatly assist other developers. Typically, simplification results in functionality decrease and there is need to find balance between functional possibilities and simplicity. The best balance can be achieved by responsibility sharing between developers providing them with different levels of simplicity.

Simplification means not only to reduce a number of actions during component creation and final application building, but it also requires a solution in interconnection between these two processes. Another words, improvement also can be done in the components management part to maximally simplify their reusability and provide separation of component developers from final application developers providing them with different levels of simplification.

In addition to this, PWAs have special concepts that are uncommon even for standard website developers. These concepts are related to the requirement to work offline and imitate native apps. To achieve maximal simplification workaround in this part also has to be done.

To sum up, there is a need to develop a tool that will allow building component-based responsive GUIs with minimum efforts but will provide enough programming level interaction using view model powered by a data binding and an option to get more customization possibilities at the cost of efforts. This tool must be a simple interface to component-based GUI frameworks. Excellent candidate as a MVVM and data binding provider is Vue.js because it is modern but very popular already. It is quite simple and lightweight compared with other frameworks and it still does not have graphical building tools for it. This tool requires ready components that must be created using more control. In the component creation area trade-off between simplicity and functionality must be done paying more attention to the functionality part. But as was said in the previous paragraphs, simplification is required not only in the final application and component building part, but also in the PWA-specific part and in the component-management part. This means that a single tool is not enough. Problem solution requires development of a small ecosystem or toolchain which will provide a simplified component and PWA building approach with centralized management of these components.

1.4 Problem relevance and research contribution

Existence of a simple and fast but still powerful way of building component-based progressive web apps and web user interfaces in general saves time for development and decreases learning requirements. This results in development efficiency increasing and money saving what is very important in the business area.

This thesis demonstrates efficiency improvements in the PWA development process compared to traditional approaches using simplification methods developed in it and conditions required for their effective use. The results of this thesis can be used in future researches in the same area for development of new approaches or for improvement of the existing ones.

1.5 Research questions

There are presented the main question of this master thesis with their direct sub-questions. In the future these questions will be decomposed to more sub-questions.

Main: How to simplify process of the building component-based progressive web apps?

RQ-1: How to simplify the process of component creation?

RQ-2: How to simplify the process of the final application building?

RQ-3: How to organize components management?

RQ-4: How to get a working PWA with minimum efforts?

1.6 Methodology

This work consists of 2 parts: In the first part a conceptual solution for the stated problem will be developed and in the second part this solution will be implemented in the form of a prototype. The developing prototype is a method itself to solve the stated problems. Its existence proofs that the stated problems are solved if its requirements were specified based on the conceptual solution and were fulfilled. This is known as a "proof by construction" in the Design Science Research [5]. The following part describes a methodology used in this thesis to achieve the final result.

The first part of the work is fully dedicated to conceptual solution development for the problems reflected in the research questions: component creation, final application building, components management and PWA producing. To achieve the result, I shall:

1. Analyse the problem area and find the parts that can be simplified.
2. Attempt to improve them desiring improvement in simplicity.

3. Evaluate found solution using analytical methods.

The conceptual solution will be developed in the second step. The corresponding research question will be decomposed into sub-questions and there will be made an attempt to find answers to them. Some sub-questions can also be decomposed to smaller ones. Solutions that will be developed for the parts found on the previous step answer these questions.

The third step analytically evaluates the found solution using the following questions: Is it makes a development of component based apps simpler and how big is the difference? What does it exchange for this simplification and how much? What are the optimal conditions to use this artefact for its optimal performance? As the solution is not implemented yet, evaluation contains approximate guesses only.

The second part of the work is fully dedicated to the implementation of the conceptual solution developed in the first part. The conceptual solution will be mapped to the requirements of the prototype. Based on these requirements will be selected tools for implementation. Also design of the prototype in the form of UML (Unified Modelling Language) diagrams will be developed based on them. After that the implementation by itself will be done to produce a working prototype. The following steps represent implementation part.:

1. Requirements specification
2. Design
3. Implementation

In the end of the implementation part will be conducted an additional evaluation phase using experimental testing methods to measure quantitative properties of simplicity such as amount of hand written code.

1.7 Thesis structure

The rest of the thesis is structured as follows. Chapter 2 provides an overview of the used technologies. It contains information important to understand the rest of the thesis if a reader is not familiar with the area of this thesis. Chapters 3, 4, 5 and 6 are related to the conceptual solution development and evaluation. The chapter 3 is dedicated to simplification in the area of component creation and the chapter 4 is dedicated to simplification in the area of final application building using these components. The chapter 5 is dedicated to solution that will connect the both: component creation and final application building. Also it shows how to make components more usable. The chapter 6 is dedicated to final steps that will make an PWA from the application. Chapters 7, 8 and 9 are related to the implementation of the conceptual solution developed in the previous chapters. Implementation of the component creation part is described in the chapter 7, implementation of the final application building is described in the chapter 8. Chapter 9 contains description of the required remote server part. Chapter 10 contains final evaluation and results description based on the implemented solution.

2 Technology and concepts overview

The following chapter contains an introduction into technologies and concepts important to understand this thesis. It gives a brief introduction to PWAs, component-based design, MVVM, data binding and Vue.js framework.

2.1 Progressive Web App

Progressive Web Applications or simply PWA is a special type of web applications introduced by Google in 2015. The web application has to fulfil number of requirements to be called a PWA [6].

- Reliable - Load instantly and never show the error screen, even in uncertain network conditions.

- Fast - Respond quickly to user interactions with silky smooth animations and no janky scrolling.

- Engaging - Feel like a natural application on the device, with an immersive user experience.

The reliability requirement means that the application has to work even without the internet. This requirement is fulfilled using service workers. Service workers is a special type of web workers (scripts that can be run in a separate thread) that can handle network requests and return the response without using the internet. This is done by caching results of requests for subsequent access. When internet connection is enabled, then the response can be fetched from the internet, but when it is disabled, then the response can be fetched from the cache. This is extremely useful for static assets like stylesheets, html pages, scripts and multimedia files. In addition to requests handling, service workers can add such possibilities like push notifications and background synchronization [7].

The requirement to be fast relates to the performance and loading time. Google advises to measure this using the RAIL (Response, Animation, Idle, Load) model. RAIL is a user-centric performance model that breaks down the user's experience into 4 key actions [8]:

- Response: Complete a transition initiated by user input within 100ms. Users spend the majority of their time waiting for sites to respond to their input, not waiting for the sites to load.
- Animation: Produce each frame in an animation in 16ms or less and aim for visual smoothness.
- Idle: Maximize idle time to increase the odds that the page responds to user input within 50ms.
- Load: deliver content and become interactive in under 5 seconds.

Engaging means that PWAs are installable and live on the user's home screen, without the need of the app store (place where users can get new applications). They offer an immersive full screen experience with help from a web application manifest file and can even re-engage users with web push notifications. This is done using a web application manifest. The web application manifest allows developer to control how his application appears and how it is launched. He can specify home screen icons, the page to load when the application is launched, screen orientation, system GUI configuration. Installation process is implemented using an installation banner what is shown when a user visits the application for the first time. After clicking the banner, the application can be added into the home screen. In contrast to the previous two requirements, engaging requires support from the operating system. In early 2018 the best support was on Android. The latest versions of this system supports a technology called WebAPK. WebAPK is a container for PWA that provides it with native application behaviour.

PWAs is a quite new concept and this is important to define a degree of their support by different browsers. The most important part is to support service workers because feature to work offline is the most noticeable one that is not available for standard web applications. For mobile devices this is also important to support web application manifest to make the application look like the native one on mobile devices. The two tables below show how modern browsers support PWAs (there is minimum version number in the middle column). Using this info can be said that PWAs are ready for widespread use.

Browser	**Version**	**Released**
Edge	17	No *
Firefox	44	Jan 27, 2016
Chrome	45	Sep 2, 2015
Safari	11.1	No *
Opera	32	Sep 17, 2015

Table 1. Service workers browser support [9]

Browser	**Version**	**Released**
Chrome (Android)	49	Mar 3, 2016
Safari (iOS)	11.1	No *

Table 2. Web application manifest support on mobile platforms [9]

* Stated version was still in development and was not released for production.

2.2 Component-based responsive web design

Today GUI of the majority of web applications is built from reusable components or modules (sometimes are also called widgets). For example, on web pages you can find such components nonstandard for the web like breadcrumbs, context menus, modal windows. Nonstandard means that they do not have their dedicated representation in the DOM (Document Object Model). A developer has to build these components using standard DOM elements and write logic for them by himself. Of course writing all from scratch gives maximal degree of freedom, but freedom and flexibility are not the same things.

In his article Dennis Kardys says: “Freedom, as it pertains to the process of assembling web pages, implies the ability to make decisions based on individual judgment. Concept of freedom never seems to work in the best interests of the site. Freedom breeds design decisions that are based on particular instances of content and context. On a small sampling of pages, this might not be so bad. But as the number of uniquely art directed pages increases, or as additional content producers begin to impose their stylistic discretion, design standardization falls by the wayside. The result is pages that look good on their own, but lack any sense of cohesion as a whole and end up negatively impacting the site's user experience (UX).” [10]. Here is a list of problems that can take place if developer uses the free form design [10]:

- Formatting that’s based on individual discretion rather than content structure leads to the inconsistent application of styles.
- Style that’s applied to elements within the content, rather than controlled globally, convolutes your content with inline HTML and CSS code. This can mess up how content is reused across your site, how it is rendered across different screen sizes, or how it can be adapted in the future.
- As the number of pages with custom layouts increases, the design appears more haphazard. This negatively impacts the user’s navigational flow and can create needless confusion.

“A flexible system, on the other hand, implies that a site has been designed with enough foresight to handle diverse author needs and content requirements. Unlike tools that empower content authors to design their own pages and layouts, flexible design systems work by providing content authors with the ability to structure page content, select modules from a library of reusable components, and apply metadata which provides instructions for making dynamic content and template formatting decisions” [10].

Modularity is the key to developing of the flexible design system. To be modular system must have interchangeable parts, and these parts are called components. In Web terms the component is just a generic term for any pre-defined object that you intend to use across multiple pages (this term also has another names like widgets or modules) [10].

Also modern GUI has to perform alternatively good on different devices. Another words GUI has to adapt to different screens. This is called responsive web design. The main challenge here is that a desktop computer screen and a mobile screen have a similar resolution but very different physical sizes. This results in different dpi (dots per inch) and hence content on mobile browsers looks unacceptably small. To overcome this browsers provide the viewport that is smaller than the native one [11]. This helps with the size, but on the other hand, after that elements do not fit in the browser window. This problem is solved using media queries. Media queries specify CSS rules that can be applied to the specific device types. A set of screens that is a target for a set of CSS rules is called a breakpoint. Presently developers do not deal with the responsive design by themselves and use one of the appropriate frameworks like Bootstrap.

2.3 MVVM and data binding

MV* patterns like MVC (Model-View-Controller), MVP (Model-View-Presenter) and MVVM (Model-View-ViewModel) was created to improve scalability, reliability, maintainability, code reusability and testability of applications [12]. The main idea of them is to separate a Model (contains business logic) from a View (defines representation for a user) using an intermediate component. These patterns are shown in the Figure 1. In case of MVC the View renders the Model and sends user input events to the Controller which calls methods of the Model. As a result, the View depends on the Model. In case of MVP there is the Presenter instead of the Controller. The View renders data provided by the Presenter and sends user input to it while the Presenter calls methods of the Model and reads its data. Unlike MVC, in MVP the View and the Model are fully separated. MVVM is quite similar with MVP. The main difference is that it adds an additional data binding layer. Like a Presenter in MVP, a ViewModel in MVVM reads Model's data and calls its methods. Changes in the ViewModel are reflected in the View due to a data binding layer. In case of one-way data binding user input events must be handled manually. In case of two-way data binding changes in the View are reflected also in the ViewModel without explicit events handling. Existence of a data binding layer decreases complexity of the ViewModel layer comparing to the Presenter.

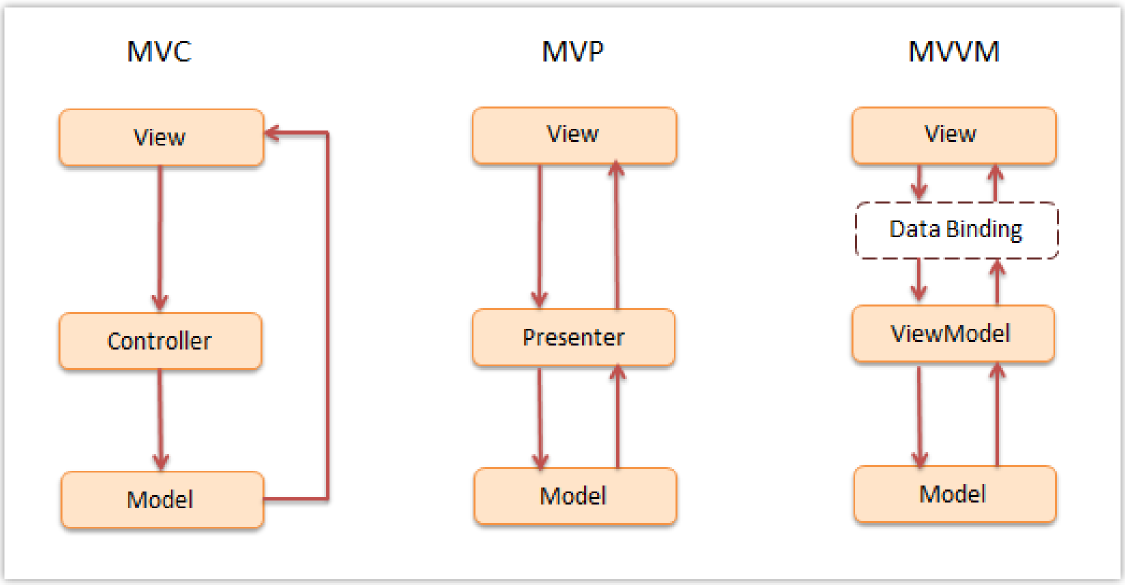


Figure 1. MV* patterns [13]

Data binding is a very important part of the MVVM pattern. It synchronizes the ViewModel with the View. Data binding itself is a type of reactivity. Reactivity in programming means that instead of assignment of static values of expressions to variables, the variables get a real expression as is. And when components of this expression are changed, then the value of the variable is also changed. There exist two types of data binding: one-way and two-way. In case of the one-way data binding, changes in the bound object are reflected when changes in the main object take place. But this does not work in an opposite direction. In case of the two-way data binding changes in both objects can be reflected in the another one.

2.4 Vue.js

Vue.js was chosen as a MVVM and data binding provider in this thesis. Vue.js or simply Vue (pronounced /vjuː/, like view) is a progressive framework for building user interfaces. Unlike other monolithic frameworks, Vue.js is designed from the ground up to be incrementally adoptable. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. On the other hand, Vue.js is also perfectly capable of powering sophisticated Single-Page Applications (SPAs) [14].

To provide MVVM Vue.js uses reactivity type based on the data binding. It uses ability of the JavaScript to define properties. A property consists of a getter and a setter method. It provides writing and reading as a variable, but these operations become calling of the getter in case of reading and calling of the setter in case of writing. For parts of the ViewModel Vue.js defines getter/setter pairs during the initialization phase. When changes in the ViewModel take place the framework can be notified of them and it can update its virtual DOM tree. The virtual DOM stores state of the Vue.js nodes that are not valid DOM nodes. Synchronization between the DOM and the virtual DOM is asynchronous relatively to changes in the virtual DOM. The virtual DOM accumulates changes that can be written to the DOM afterwards. This process is shown in the Figure 2. Vue.js focuses on the one-way data binding, it synchronizes the DOM with the ViewModel. To provide a reversed synchronization events can be used. An exception is input field that is provided with the two-way data binding by Vue.js out of the box.

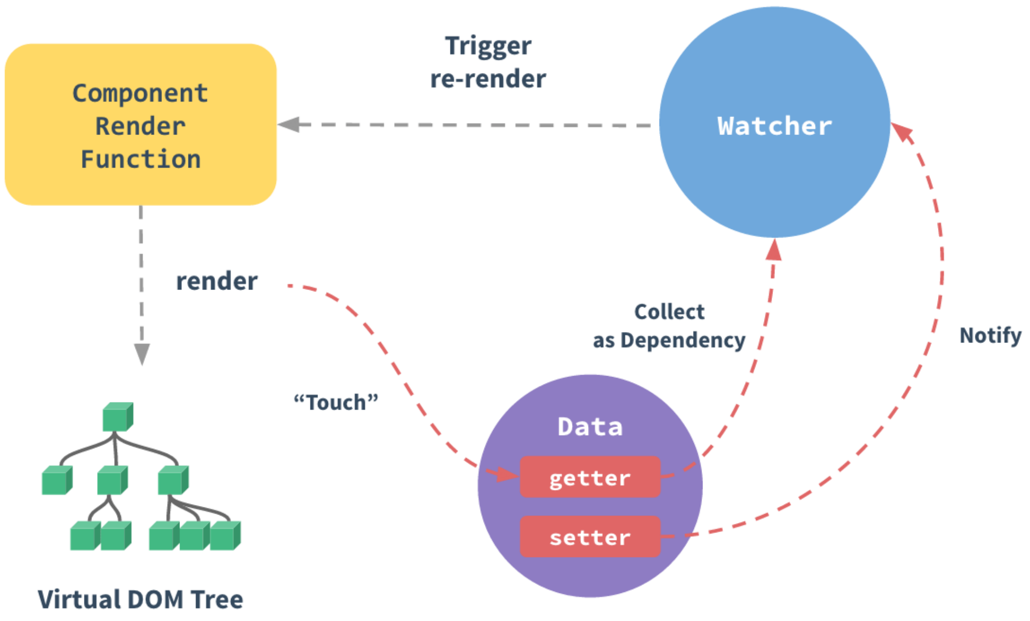


Figure 2. Data binding in Vue.js [14]

Vue.js by itself does not have support for such important PWA concepts like client-side routing, centralized state management. But it provides official extensions: vue-router for client-side routing and vuex for centralized state management. Client-side routing is a core concept for single page applications (PWAs are also SPAs), it means to display another content or another screen (another page in terms of the convenient routing) without refreshing a browser. Centralized state management becomes important for large scale applications when multiple pieces of state are scattered across many components and interaction between them becomes complex [14].

In this paragraph is given a brief comparison with other 2 the most popular frameworks: React and AngularJS. Both Vue.js and React utilize a virtual DOM, provide reactive and composable components, maintain focus in the core library, with concerns such as routing and global state management handled by companion libraries. React has notably larger ecosystem that Vue.js because it is older and more popular, but Vue.js ecosystem also continues to grow actively. Performance areas Vue.js can be better because it tracks dependencies during its rendering process, so the system knows precisely which components actually need to re-render when state changes. In contrast, in React when a component’s state changes, it triggers the re-render of the entire component sub-tree, starting at that component as root. In React, everything is just JavaScript. Not only are HTML structures expressed via JSX (JavaScript Syntax Extension), a declarative XML-like syntax that works within JavaScript. The recent trends also tend to put CSS management inside JavaScript as well. Vue.js embraces classic web technologies and builds on top of them. In React, all components express their UI within render functions using JSX. In Vue.js, there are also render functions and even JSX support, however, the most convenient way is to use templates. React is renowned for its steep learning curve. Before you can really get started, you need to know about JSX and probably ES2015+ (ECMAScript 6 and higher). You also have to learn about build systems. On the other hand, in case of Vue.js you only have to include a single script tag and then you can start writing code. If compare AngularJS and Vue.js, then they have quite similar syntax. AngularJS was an inspiration for Vue.js developers. When they were creating their framework, they wanted to make it much simpler in terms of API and design. AngularJS has strong opinions about how your applications should be structured, while Vue.js is a more flexible, modular solution. AngularJS uses two-way binding between scopes, while Vue.js enforces a one-way data flow between components. Vue.js has a clearer separation between directives and components while in AngularJS directives do everything and components are just a specific kind of directive. Vue.js has better performance and is much, much easier to optimize because it doesn’t use dirty checking. Angular 2 (Formerly known as Angular 2) has fixed some of drawbacks that was in AngularJS. It is deeply integrated with TypeScript and requires using of it. It has noticeably better performance that his predecessor and can compete with React and Vue.js. Vue.js is much less opinionated than Angular 2, offering official support for a variety of build systems, with no restrictions on how you structure your application. Also its learning curve is much steeper. Both React and AngularJS have larger size than Vue.js [14].

2.5 Definition of simplicity

The main aim of this thesis is to simplify process of component-based PWA building using Vue.js as a MVVM provider. But what this simplicity means? This word was already used quite often and also will be used often in next chapters. To continue, it is very important to define what simplicity means.

In his book John Maeda defines 10 laws of simplicity [15]. Some of them can be used to define what is simplicity in this thesis. Their application is shown in the table below.

Law	**Application**
Reduce: The simplest way to achieve simplicity is through thoughtful reduction.	Remove functionality that is not required often and focus on functionality that developers require maximally often.
Organize: Organization makes a system of many appear fewer.	Help developers with the management of their assets. Allow them to interact only with things that really affect the final result hiding all the rest. Also allow them to focus on the concrete part with possibility to hide all the rest and make these parts easily accessible.
Time: Savings in time feel like simplicity.	Increase development speed maximally reducing the number of required actions. Development speed is inversely proportional to time. And time depends on amount of required code. As a result to increate development speed automatic code generation must be used where it is possible.
Learn: Knowledge makes everything simpler.	Decrease learning requirements needed to start using the solution developed in this thesis. Replace some concepts with more intuitive ones if it is possible.
Differences: Simplicity and complexity need each other.	Find a compromise between simplicity and functionality. The user of solution must not feel that he is severely constrained in functional possibilities. Give a possibility to achieve more functionality by the cost of simplicity.

Table 3. The definition of simplicity

3 Simplifying component creation

In this chapter will be developed a conceptual solution with the aim to the simplify process of Vue.js components creation. The traditional way will be analysed to find the parts that can be simplified. After that will be attempted to improve them orienting to the improvement of simplicity. The developed solution will be evaluated using analytical methods.

Research questions that will be answered in this chapter:

RQ-1: How to simplify the process of component creation?

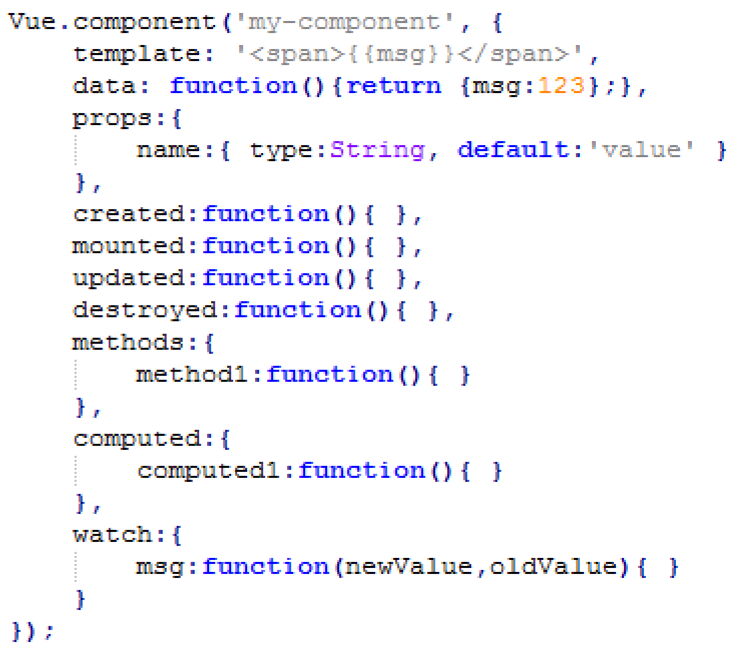
RQ-1.1: How to simplify Vue.js concepts?

RQ-1.2: How to map between simplified concepts and the real world?

RQ-1.3: How to test the component?

3.1 Overall structure

Vue.js defines its ViewModel instance in the form of a JavaScript object. A component itself is a special type of a Vue.js instance that was designed to be reusable. To be available, the component must be registered globally or locally. The global registration makes the component available everywhere while the local one makes it available only within another component. Component has fields that represent its concepts. Example of these fields are hooks, methods, computed, watch, data, template. Not all fields are mandatory, but names of the fields are always the same. As a result, developer must write always the same repetitive code. The amount of repetitive code depends on the complexity (a number of used fields). Moreover, developer must know names of these fields and signature of the value. The figure below shows code that illustrates repetitive parts of the component.



```
Vue.component('my-component', {
    template: '<span>{{msg}}</span>',
    data: function(){return {msg:123};},
    props:{
        name:{ type:String, default:'value' }
    },
    created:function(){ },
    mounted:function(){ },
    updated:function(){ },
    destroyed:function(){ },
    methods:{
        method1:function(){ }
    },
    computed:{
        computed1:function(){ }
    },
    watch:{
        msg:function(newValue,oldValue){ }
    }
});
```

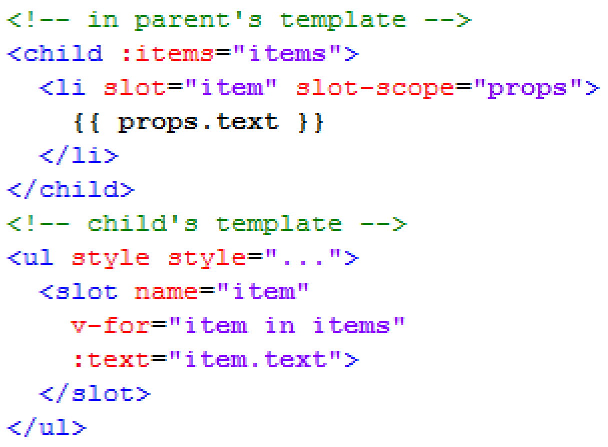
Figure 3. A Vue.js component example

An obvious solution is to generate a skeleton of the component with component registration code. Then the developer has to fill the gaps of the skeleton with his code. There exist numerous code generators for Vue.js components like vue-generator (https://github.com/hjeti/vue-generator). As this thesis is targeted to simplification this leads to the question: What interface provide to the developer that will hide all unimportant parts and will assure enough flexibility. Also it has to minimize the number of actions to achieve the result.

The solution is to provide only labelled input fields where the developer will write his code. These fields must be comfortable for writing programming or mark-up code depending on the filed type. These fields must have basic properties of code editors like syntax highlighting. For example, template field must provide syntax highlighting suitable for HTML code while hook field must provide JavaScript syntax highlighting.

3.2 Template

A template defines appearance of Vue.js components and relates to the View part. After processing it becomes a DOM node that will be shown to the user. The template supports special Vue.js directives that are not part of the HTML and are ignored by browsers. These directives are used for such things as list rendering, condition rendering, data bindings and code embedding. The template must have a root node, otherwise it cannot be processed. These aspects require some knowledge from the developer and can become the pitfalls for the new coming ones. The template can be styled with CSS, but as they are embedded into the body of the document, the only one way to style them is to use inline CSS. Inline CSS has some disadvantages like a requirement to apply it to every element that must be styled and impossibility of styling pseudo-elements [16]. Vue.js also introduces the concept of slots. This is a template that can be provided by a parent of the component. Slots can have names and default values defined by the component. The figure below shows the template code that illustrates some Vue.js directives and the child-parent interaction using slots.



```
<!-- in parent's template -->
<child :items="items">
  <li slot="item" slot-scope="props">
    {{ props.text }}
  </li>
</child>
<!-- child's template -->
<ul style style="...">
  <slot name="item"
    v-for="item in items"
    :text="item.text">
  </slot>
</ul>
```

Figure 4. A Vue.js templates example

Template and slots both are pure code and it is very hard to simplify something there without severe flexibility loss. But there are some things that still can be done here. Some hints for Vue.js directives can be provided in the template’s editing field to help new developers. Requirement to have a root component can be omitted by providing a default one (more info in the next chapter). Also slots can be extracted from the template and shown to the developer without a need to open the template editing field. The problem of lack of inline CSS can be solved using a separate code field for it with an appropriate code highlighting mode. This will also result in a better separation of concerns [17].

3.3 Functions

Vue.js uses functions for the following concepts: hooks, methods, computed properties (computed field). Figure 3 has an example of these functions with an empty body. Hooks are functions that are called by the framework during component's lifetime. There are 4 of them: created, mounted, updated, destroyed. The number of methods and computed properties can vary. Methods are very similar with computed properties. The difference is that computed properties cache the result they return. All of 3 function types can emit custom events to parent. Events are not declared explicitly and can be fired using the $emit method of the component.

As in the case with templates hooks, methods and computed properties are pure code and simplification will result in notable flexibility loss. As slots can be extracted from templates, events can be extracted from hooks, methods and computed properties. This will help developer to have a full idea about events that his component emits. Both methods and computed properties can be created and removed. To assist these processes and degrease amount of code editing they can be grouped into lists with the "add" and "delete" buttons.

Another idea was to transform the concept of hooks into the concept of constructor/destructor similar with the C++ OOP using the "created" hook as a constructor and the "destroyed" hook as a destructor. This would be familiar for the developers that know C++. But the number of those is not so big [18]. Moreover, this requires to drop support of 2 hooks that could affect functional capabilities in some cases like a possibility to react to virtual DOM changes. Based on these arguments this idea was rejected.

Another rejected idea is to exclude computed properties as they can fully be replaced by functions. Of course this will result in lack of caching possibilities, but functionality will not be affected. A decision to reject this idea was made based on the requirement to separate the concept of methods that change state of the component and the concept of computed properties that produces non-trivial value based on the current state. Lack of computed properties that behave like normal properties would affect readability.

3.4 ViewModel state

A source of data in the ViewModel of Vue.js are properties (the term of Vue.js) and data fields. Both are serviced by the data binding provided by the framework. The difference is that the property value comes from the parent component, while data represents inner state of the component. As shows in the Figure 3, the data field is a function that returns the real data object. This is not intuitive and can be another pitfall for unexperienced developers. Properties allow type and default value specification, but do this in a clumsy way. Vue.js also allows to define watchers for component data. They are functions that are called when the data is changed.

Ultimate workaround here is to fully omit the data field and generate it automatically. But how to decide what to put into it? The answer can be found in the ES6 (ECMAScript 6) and Python class constructors. Class fields in these languages are not defined as a part of class, but assigned inside a constructor. Using this approach, all methods, hooks and computed properties can be searched for similar assignment syntax. After that the found names can be used to generate a data field with initially undefined values. Using this approach will free developer from knowing anything about the data field. It will have reactive data using the familiar way.

As each data entry can have only one watcher, then a list of watchers can be created automatically. This also solves the problem of visual data representation. There is no need to create a separate list like in the case of events, all data entries will be displayed in the list of watchers. As signatures of all watcher methods are the same, then they can be generated automatically.

Another significant improvement can be done in the case of properties. In typed languages like Java variables are defined as a type-name-value triplet. This approach can be used to simplify definition of properties. A special input field with own highlighting mode can be provided to specify properties in the type-name-value style that will be transformed into native property syntax. The type and the value parts are not mandatory. The value part is a default value of the property.

But can properties and data be merged together as they both represent a data source? This would lead to more simplification. But this is impossible because the data field was designed to represent the inner state of the component and to be changed by it, while properties was designed to pass data from the parent component to his child and those concepts cannot be mixed.

3.5 Two-way data binding

Vue.js provides only a one-way data binding out of the box. Only one exception from this rule is the v-model directive that enables two-way data binding for input HTML nodes. For arbitrary components this directive can also be used, but an event with appropriate signature also must be fired when component state is changed. This approach handles changes in the entire component state. This means that if only one variable in the component state is changed, then the parameter of the event handler contains entirely new component state anyway.

As was described before, properties are defined separately and must not be changed by component itself. But in case of two-way data binding component wants to change them and this would be intuitive. This can be done by analysing code for property assignments and replacing these entries with event emitting. Instead of property assignment, a temporary variable can be created and the result of the expression can be assigned to it. After that an event can be emitted that contains the value of this temporary variable. In contrast to v-bind directive this approach allows to bind parts of the component state separately. To use this approach, the application builder that uses the component must also support additional functionality that will be described in the next chapter.

3.6 Preview

In the traditional approach when there is need to test the component, a HTML page must be created with an included Vue.js library, a simple body layout and the component itself. Then this page must be opened in a browser. After making changes in its code, the page must be refreshed to apply these changes.

These actions can be skipped if development and execution environments are the same. As Vue.js is a JavaScript front-end framework then development must also take place in the browser for the best result. When some parts of the component were changed then processing of the changed parts can be triggered. The processing is different for different parts. For example, changes in the template will result in update of the slot list, while changes in methods will not. But how to decide when to trigger the update process? It is obvious that to update preview after each code change is too expensive. But if editing fields can be opened and closed, then the system will know when the developer has finished editing and the preview can be updated. If there was an error in the code, then an exception can be handled and the source of the error can be displayed in the preview frame. Handlers can be added for both: Vue.js runtime errors and JavaScript syntax errors.

3.7 Evaluation

The next table contains the analytical evaluation of the developed conceptual solution for the component creation simplification. The result column contains pros of the solution. It can also contain cons. In this case the optimal conditions are given.

Solution	**Result**
Generate component skeleton code providing editing fields only for required parts that define the View and the ViewModel of the component.	Pros: Amount of handwritten code decreases. The developer focuses only on the required parts. Frees developer from knowing the automatically generated parts that reduces learning requirements and error probability. Cons: Some special features of the Vue.js like custom directives become inaccessible. Optimal conditions: The developer wants to focus only on the component's View and ViewModel using only standard features of Vue.js.
Generate a root node of the template.	Pros: Eliminates the mandatory requirement to have a root node. Possibility to automatically adapt the root node to use with a concrete layout solution. Cons: Needs manual code editing in case of the intent to use the generated component somewhere else. Optimal conditions: Use the component creator in tandem with the application builder developed in this thesis.
Separate the style from the template.	Pros: Possibility to use pseudo-elements. Better separation of concerns and organization. Visually more readable and easier to follow.
Provide a list of slots and events.	Pros: Gives developer a clear idea what slots and events component has without searching them inside the code.
Group methods and computed properties into list with add/remove/edit possibilities.	Pros: Reduces amount of code editing actions that must be taken to create or delete a function. Lists help with organization.

Provide custom brief syntax for property definition.	Pros: Provides a clearer and brief way to define properties, as a result error probability and amount of code decreases. Cons: Cannot use custom validating functions Optimal condition: There is no strict requirement to filter parent input besides type control.
Generate the data field automatically by analysing code of the component for traditional class fields initialization syntax.	Pros: Eliminates a mandatory requirement to explicitly define the data field that reduces amount of handwritten code, learning requirements, error probability and pitfalls.
Automatically generate an editable list of watchers for data entries.	Pros: Automatically synchronizes the list of watchers with the data. No need to define watchers manually. Generates watcher's signature that reduces amount of handwritten code and error probability.
Generate events in place where the component changes its properties.	Pros: Allows to use two-way data binding without writing events explicitly. This allows more brief, intuitive and clear syntax.
Provide an automatically updating preview with possibilities to show errors and exceptions.	Pros: No need to write a html page and refresh the browser after editing manually. Shows errors and exceptions inside the preview without need to open the console window.

Table 4. Component creation evaluation

4 Simplifying application building

In this chapter will be developed a conceptual solution with the aim to simplify the process of final application building. The traditional way will be analysed to figure out how to improve this process orienting to the improvement of simplicity. The developed solution will be evaluated using analytical methods.

Research questions that will be answered in this chapter:

RQ-2: How to simplify the process of the final application building?

RQ-2.1: How to simplify a GUI layout?

RQ-2.2: How to provide different SPA screens?

RQ-2.3: How to make a GUI responsive with minimum efforts?

RQ-2.4: How to simplify interaction between the application and its components?

RQ-2.5: How to customize components?

RQ-2.6: How to simplify the ViewModel of the application?

RQ-2.7: How to test the application?

4.1 GUI layout

Web browsers provide reach possibilities to a define layout of GUI elements. Typically, it is not enough to define layout using HTML only and additionally requires application of CSS styles. But these reach possibilities typically result in frustration for new developers as there exist different ways to achieve same results. One example is the display property that changes standard behaviour of the styled element and can make inline elements from block and vice versa. There exist GUI frameworks that use own abstraction to provide one understandable way to handle GUI layout of the application. The most popular and considered as the best [19] of those frameworks is Bootstrap. It uses rows and columns abstraction by applying appropriate CSS classes. Developer uses a top level container where it puts rows. The row uses entire parent’s width. The row can contain up to 12 columns. Columns has discrete size (in 1/12 of parent width) and contain content or other rows. This system is named Bootstrap Grid. In addition to this, Bootstrap grid allows such features as alignment, column offsets and custom column order. Another benefit of using Bootstrap Grid is that it has multiple variants of each class for different breakpoints to make layout responsive. The Figure 5 shows a layout example using Bootstrap Grid with 3 rows and columns targeted for different breakpoints. But still the developer has to edit HTML code manually as Bootstrap Grid provides only a set of CSS classes. Also styling with classes is not as intuitive as styling with properties. Another problem is that Bootstrap Grid deals only with "flat" GUI layouts. This means that it works inside one screen (in SPA terms) and does not deal with a stack of these screens what is important if a HTML page contains multiple screens of the same application (what SPAs typically are).

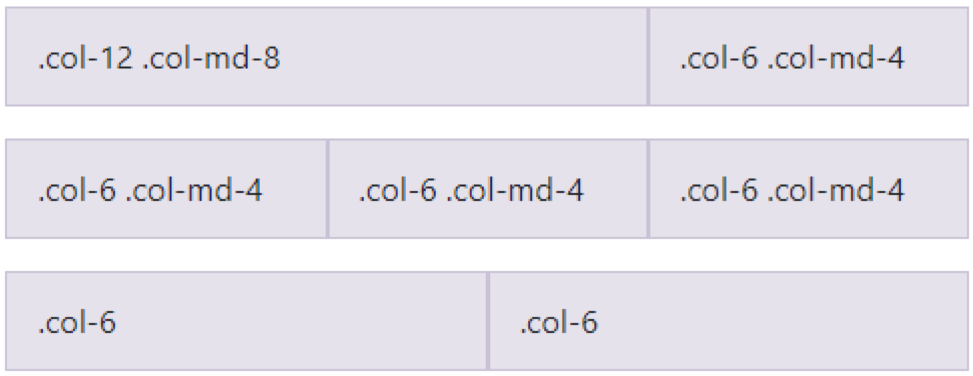


Figure 5. A Bootstrap Grid example

The figure above demonstrates that Bootstrap Grid can be easily visualized using rectangles that represent column content. This visualization is also very clear and easy to understand. Then it is excellent starting point to make GUI layout builder. This visualization can be made interactive by adding possibilities to add, edit or remove content. Another words, GUI layout building can be organized entirely in the graphical way.

As Vue.js components have a root element, then this element can be used to make the component compatible with the previously described Bootstrap Grid system. To do so, this root element must have classes that will make a Bootstrap Grid column from it. It has to have properties that define its width and offset in columns, order and alignment. These properties define layout of the application and do not have to be reactive. To correspond

Bootstrap Grid framework these properties must be converted into appropriate classes using code generation.

To deal with the responsive layout requirement there is need to provide multiple sets of the previously described layout properties. Bootstrap Grid has 5 breakpoints that can be a little overheating. We typically talk about desktop and mobile apps. This requires to use only 2 breakpoints that is entirely suits for this thesis as it focuses on simplification. But then what maximum screen width (actually viewport width) the device has to have to be treated as a mobile one? Things are quite clear for smartphones and desktop screens, but tricky for tablets [20]. Instead of forcing developer to use predefined value this can be provided by himself. But on the other hand he obviously wants to support all devices and do not know this answer also. The simplest decision is to use the same breakpoints as Bootstrap uses.

Requirement to embed rows inside another rows means that rows must have similar layout parameters as components. Another words, they also have to be a type of Bootstrap Grid column by themselves and can be treated as a predefined type of Vue.js component. A top level Bootstrap Grid element is a container. Container contains rows and rows contain columns. Container cannot contain columns directly. Bootstrap Grid has 2 types of containers. The standard container has discrete and constrained width while the fluid one expands smoothly and uses all available width. The fluid container better suits for web applications as it uses available width more effectively.

By far was described a layout that was called a “flat” one. But what to do if there is need to make a modal window or provide a fixed menu? Another example is multiple screens of SPA where only one of them has to be shown to a user. These examples need a set of containers with different display properties. Modals and fixed menu must be displayed as fixed while screens must allow to scroll their content. To deal with this requirement can be used the layer abstraction. This abstraction is quite familiar from graphical editors like Gimp and Photoshop. The main idea is to provide another custom abstraction term called layer to the existing row and column terms. The layer is a container for Bootstrap Grid system that makes it a holder for entire Bootstrap Grid layout. Layer can be fixed (position is relative to the viewport) or absolute (position is relative to the document). The figure below demonstrates GUI composed from 3 layers: fixed menu, content and modal window.

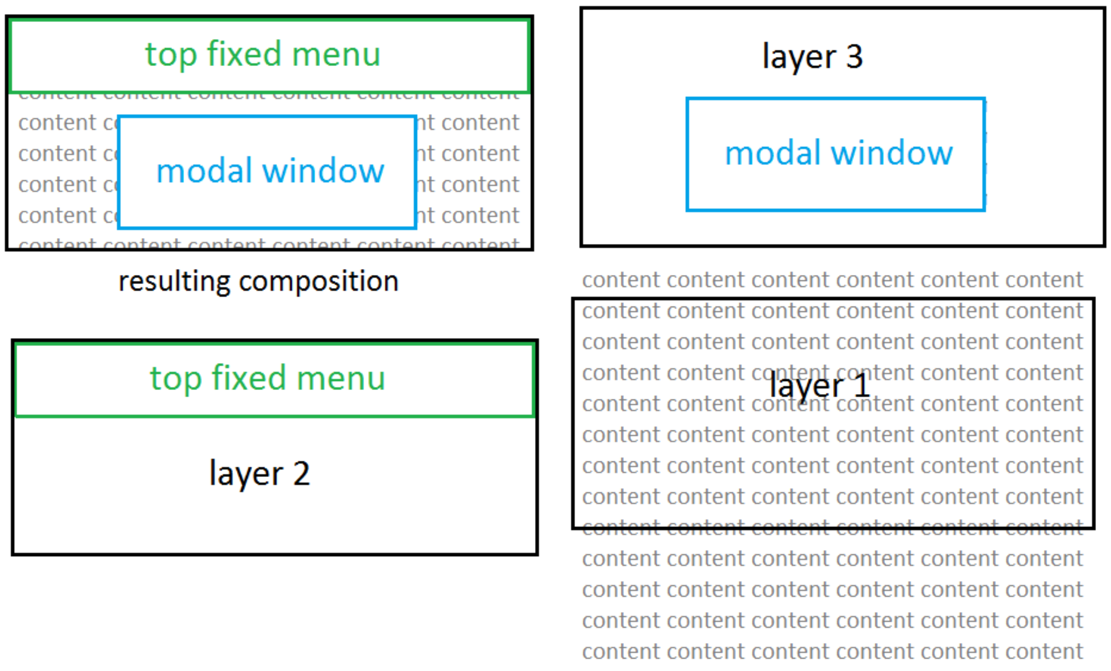


Figure 6. Composition using layers

Possibility to enable/disable layers is very important for SPAs. Each screen of the SPA can be putted to a separate layer and routing can be made via showing/hiding these layers. Alternative possibility is also important for rows and components to show/hide parts of the GUI. This can be made using conditional rendering provided by Vue.js. This requires that layers locate inside Vue.js scope. A reactive display property can be provided to the developer that will be transformed into the Vue.js conditional rendering under the hood.

To sum up, the entire process of the final application building can be done in the following way: The developer adds layers by clicking "add" button and giving them a name. Then it can select a layer and add some rows to it also by clicking "add" button and giving a name for future referencing from the code. Similarly, components or other rows can be added into the selected row. Selected elements (layers, rows, components) can be removed by clicking "delete" button or selected for editing. This building process creates component hierarchy that can be processed by code generators to create the resulting application.

4.2 Component customization

The only one traditional way to customize a component in Vue.js is to use slots. They allow parent to define part of the component’s template. If parent does not specify slot content, then a default one will be used that was defined inside the child. The developer has to know the name of the slot and its syntax to effectively use it. He also cannot use CSS style to customize the component as Vue.js template supports only inline styling as was shown in the previous chapter.

A style problem can be solved by providing the developer with a style editing input field with CSS-oriented syntax highlighting. By default, it loads style provided by the component developer. But style in these fields uses class selectors that makes impossible to apply unique style to the concrete component instance. This can be solved using code generation by providing CSS id for each component instance and use compound id-class CSS selectors.

The slot workaround is to provide a list of all declared slots to free a developer from explicitly knowing them. The developer can simply click a needed one and edit it in a HTML-oriented editing field. Similar with styles, this field loads the slot content provided by the component developer as a starting point for future customization by default.

4.3 Data exchange with components

The main way to pass the data into the component is to use properties. This binds parent’s ViewModel with the component forcing it dynamically display parent’s ViewModel changes. Events are the main way to get the data from the component. They are the key for two-way data binding. In traditional way, to bind the data to the property or to bind the event handler (function) to the event, the developer needs to use appropriate vue.js directives as a component’s attribute. This requires the knowledge of all methods and events that the component has.

The solution that was developed in the previous chapter allows to define property types. To assist the developer, he can be provided with the list of all properties with their types. As only the data field can be bound, then there is no need to additionally define something. The developer can simply select from a drop-down list what data of the parent ViewModel bind to it.

In the case of two-way data bound properties that was described in the previous chapter an event handler can be generated to react to their changes. This event handler simply assigns the value that is given by its argument to the appropriate data field.

The situation with events can be handled analogically. The developer can be provided with a list of events. Then he can simply select which method to use as the event handler for concrete event. But this can be simplified even more. Event handlers can be provided with correct syntax beforehand and the only one thing the developer has to do is to fill the event handler with his code.

4.4 Accessing from code

Sometimes the component must be accessed from code. Example of this situation is when there is need to use its functions such as methods or computed properties. And again, to use them effectively the developer must know them. Moreover, Vue.js does not provide simple children access from the parent. It provides the $refs property that allows to access children by the name explicitly specified as a special component attribute. This can be quite unintuitive and uncomfortable for new developers.

There was discussed the requirement to have an id to use dedicated style for the concrete component instance. This requires to give the names for all component instances. This name can be used as a name in the $refs property. Moreover, accessing to the children components can be organized by creating appropriate fields in the ViewModel. The names of these fields can be the same as the names in the $refs property. This can be done by dynamically assigning values from the $refs property to the ViewModel during component creation. This will allow accessing children without dealing with the $refs stuff.

4.5 Global ViewModel of the application

Vue.js can provide a ViewModel for the component or simply for the arbitrary HTML node. There is no much difference between these two cases. The only one difference is that in the case of the root ViewModel there is no need to communicate with the parent. This reduces number of Vue.js concepts that can be affected in this case. Also template is generated automatically and as a result only function-like concepts are left.

To get maximum from the Vue.js data binding, the root HTML node must be the node that contains all other components. This is a parent node for all layers and the corresponding ViewModel relates to the whole application. This global ViewModel is a single source of data for all components of the entire application. The global ViewModel can have only function-like concepts such as hooks, methods, computed and watchers. They can be simplified exactly the same way as in the case with single component described in the previous chapter. These simplifications include: skeleton code generation; data field generation; assisting with grouping, functions creation and deletion.

4.6 Preview

In the traditional approach, the final application is tested in the browser. After making changes the browser must be refreshed to display these changes. Also browser shows only the final app, without simplified and clear layout view where location of all elements is very clear.

As was shown in the previous chapter, if development and execution environments are the same then this gives additional capabilities. As Vue.js is a JavaScript front-end framework, then development must also take place in the browser for the best result. So far there was described the simplified view of the application layout that shows what layers, rows and components the application has. This view can be made more demonstrative if it will use Bootstrap Grid by itself to correctly match layout properties. Instead of the real components it can show only the names of them. As a result, it can correctly show width, offset, alignment and order properties. As a height placeholder a configurable fixed value can be used. Layers can be displayed as tabs. By clicking this tab, the layer can be selected and the content of the tab view can be updated to show the content of the selected layer.

The previously described simplified view does not need code generation and can be used after each layout change. But it does nod display the final application as is. To display the final result, iframe can be used as an additional preview to render this result. But this requires heavy processing and hence cannot be used after each change. It can be triggered by simply pressing a “refresh” button. Additional iframe can be used to represent a mobile screen.

4.7 Evaluation

The next table contains the analytical evaluation of the developed conceptual solution for application building simplification. The result column contains pros of the solution. It can also contain cons. In this case the optimal conditions are given.

Solution	**Result**
Provide an interactive layout view consisting of tabs (layers) , rows and columns (other rows or components) with possibilities to add/delete rows and add/delete components.	Pros: Easy and intuitive Bootstrap Grid layout composition without code writing Cons: Layout possibilities are limited by the Bootstrap Grid layout system. Optimal conditions: There is no requirement to use a complex and highly configurable layout.
Provide layout properties such as width, offset, order and align for rows and components.	Pros: Configure Bootstrap Grid layout system using only dropdowns without code writing. Cons: Layout possibilities are limited by the Bootstrap Grid layout system. Optimal conditions: There is no requirement to use a complex and highly configurable layout.
Use 2 sets of layout properties: for the desktop and for the mobile.	Pros: Possibility to use one layout for the desktop and another one for the mobile using only grid layout parameters without creating another view hierarchy. No need to write code. Cons: Only 2 breakpoints. Optimal conditions: Application does not have special layout requirements for intermediate devices such as tablets.
Extend the standard grid with the layers abstraction.	Pros: Uses very straightforward concept of layers. Layers abstraction allows using widgets with fixed positions like fixed menus and modal windows. It also allows using multiple SPA screens what is important for client side routing. They can be used entirely in the graphical mode without code writing. Cons: Very simple concept, can only be used as a direct layout layer under a root (rows cannot contain layers). Optimal conditions: There is no requirement to use complex and highly configurable layout.

Provide layers, rows and components with the reactive display property.	Pros: Developer can dynamically control visibility of any layer, row or component without explicit definition of this functionality. Frees developer from alternative code writing. Cons: Cannot use animation. Optimal conditions: No special animation requirements.
Provide a separate style editing field.	Pros: Possibility to use pseudo-elements. Better separation of concerns. Visually more readable and easier to follow. Can style every component instance without explicit id.
Provide a list of slots with the possibility to edit them.	Pros: The developer has a clear picture about component slots and their default content. He can use the default content as a starting point for customization and focus on it without dealing with the enclosing slot syntax.
Show properties with additional info such as type. Bind data by simply selecting it from a drop down list.	Pros: Provides the developer with more info about properties and frees him entirely from code writing. Allows two-way data binding.
Automatically generate empty event handlers and show a list of them.	Pros: The developer has a clear picture about component events and focuses only on the handler body. This reduces amount of hand-written code and error probability. Cons: Cannot use the same handler for different events without code duplication. Optimal conditions: All events require dedicated event handlers.
Inject children components directly into the parent's ViewModel using the name of the component instance as a field name.	Pros: More convenient and intuitive way to access components without dealing with the specific Vue.js syntax. As a result, error probability also decreases.
Show a list of component's methods and computed properties.	Pros: Gives the developer a clear picture about what component's methods and computed properties he can use in his code.
Use a global application ViewModel with hooks, methods, computed and watchers. Handle these concepts as was described in the previous chapter.	Pros: The global ViewModel acts like a centralized data store for the whole application. Hooks, methods, computed and watchers were discussed in the previous chapter and hence related result can be found in the Table 4.

Use double preview mode: The automatically updating interactive layout preview and the manually updating final preview.	Pros: The developer has a clear idea about his application layout without additional actions. If he wants to see the final result the only one action he needs is to press the “update” button.

Table 5. Application builder evaluation

5 Simplifying components management

In this chapter will be developed a conceptual solution with the aim to simplify the process of components management. Components management connects the processes of component creation and the final application building and its existence is required for proper functionality of them. Also the traditional way will be analysed to figure out how to organize this process orienting to simplicity. The developed solution will be evaluated using analytical methods.

Research questions that will be answered in this chapter:

RQ-3: How to organize components management?

RQ-3.1: How to make components reusable?

RQ-3.2: How to reduce redundancy in case of similar components?

RQ-3.3: How to make components usable in the another environment?

5.1 Component library

In the standard approach, Vue.js components are objects defined in JavaScript files. Using bundlers like Webpack (https://webpack.js.org/) with appropriate loaders help to store style and template separately in the same file. This simply a text file and it can be edited in every text editor. To share work with other developers, standard tools like version control systems or package managers can be used. Integrated development runtimes assist with these actions. As was said before, the best results can be achieved when the development environment and the execution environment are the same. But browsers do not allow to access the file system. Also the simplified approach described in the previous chapters needs storing additional metadata that is not a part of standard Vue.js components.

A browser application does not have access to the file system, but they still can save information to the non-volatile memory. During the component creation described in the chapter 3 the developer produces numerous text entities that are processed to extract additional information that helps with the development process. Example of this information is the list of events. During component editing changes in code are tracked to keep this list in the actual state. The same metadata is also needed during the application building phase. One option is to save component without this additional information by keeping content of the text fields and reprocessing it again during component loading. Another option is to save with this information. The first option requires a little bit more computation while the second requires a little bit more space. This can be done locally or remotely. Storing locally do not require the internet connection while storing remotely allows accessing from multiple devices and as a result it allows multiple developers to cooperate. But a more flexible option is to use a hybrid approach with the possibility to choose between the local storage or the remote one.

The loading/storing process can be done in following way. After editing the developer clicks the “save” button, enters the component’s name and selects where to save it (locally or remotely). The system uses the entered name as an identifier. To load the component, the developer clicks the “open” button and selects the component from a list. The list can allow searching and grouping local and remote components for better usability. During the application building phase the developer selects components to use from the same list. This approach can be named as the component library

5.2 Forking

Sometimes the developer wants to create a component based on another component. Similar concepts are called inheritance in OOP or forking in the UNIX world. Maybe the developer wants to change the component template and some methods only. One option is to make a copy of the component and edit its needed parts only. Another option is to use the concept of Vue.js called mixins. Mixins allow to specify components that will be used as parent components. The framework simply copies fields from these component into the current one. Mixins help with code redundancy and readability in case of standard approach but require knowledge of additional syntax.

The graphical approach with possibility to hide unneeded parts that was described before helps with readability and in addition to this gives more info about component parts such as methods and computed properties. Inheritance or forking can be done by simply opening and saving with another name. But this requires developer to invent a naming convention for child components and additional actions. Instead of this, the component can be opened in the forked mode. This mode makes a copy of the component and memorizes the original one. After that the developer can make its changes without changing the original component and inheritance chain will be tracked.

5.3 Export

Using the traditional approach developers create component directly for the browser or for the concrete building system like Webpack. In the first case the created component can be used directly in the browser. In the second case the component file has more comfortable structure but has to be processed by the building system. Syntax of files in these cases is different and requires changes to transform one to another.

Saving and loading that was described before work with the special component structure. But what if the developer wants to use created component outside the developing building tool? This requires to generate standard Vue.js component code and allow the developer to download this code. Different exporters can be created to provide different results for different targets. For example, code can be generated for standard ES6 module syntax (.js file extension) to use in browsers, or for Webpack's Vue.js loader (.vue file extension).

5.4 External files inclusion

Very often developers use side libraries in their project instead of writing own implementations. These libraries can also include CSS styles. In the standard approach to include an external JavaScript file the developer has to specify a script tag. To include a CSS file he has to specify a link tag.

The solution developed in this thesis allows to write both CSS styles and JavaScript scripts but for now it does not allow to include external assets. An example of these assets can be the Model part of the application that was already developed by other developers. To maximally reduce amount of hand written code the developer can be provided with an input field where it can write a list of URLs of needed assets. But how to distinguish different asset types? This can be done by analysing file extension of the URL. But if the URL does not contain a file extension then a request can be made to analyse the mime type of the response.

5.5 Evaluation

The next table contains the analytical evaluation of the developed conceptual solution for components management simplification. The result column contains pros of the solution. It can also contain cons. In this case the optimal conditions are given.

Solution	**Result**
Provide the developer with the component library where he can save his components and open them for editing or using in future.	Pros: The developer always has an easy-accessible list of components and can use/open/save them "in one click".
Provide a checkbox for storing the component remotely on the server. Display a list of remote components along with the local component library.	Pros: Allows multiple developers to share their component libraries. This makes components accessible from multiple devices and helps with responsibility sharing among developers. This simplifies work of the single developer as he can focus only on his part. Cons: All shared components are accessible to everyone. Optimal conditions: All users of the remote component library are trusted.
Possibility to fork (extend or inherit) components.	Pros: Components can be developed based on other components what decreases redundancy. Mixins functionality of Vue.js without writing additional lines of code. Cons: Cannot use multiple inheritance (when the component has more than one direct parent). Optimal conditions: Child component is never based on more than one parent component.
Possibility to export into text files in different formats.	Pros: The created component can be used outside the developing ecosystem that increases the application area.
Possibility to include external assets by specifying URLs only.	Pros: Allows to use already written assets without writing HTML tags explicitly.

Table 6. Components management evaluation

6 Simplifying transformation to PWA

In this chapter will be developed a conceptual solution with the aim to simplify the final step of the PWA building. This chapter focuses on parts specific for PWAs. The traditional way will be analysed to figure out how to improve this process orienting to the improvement of simplicity. The developed solution will be evaluated using analytical methods.

Research questions that will be answered in this chapter:

RQ-4: How to get a working PWA with minimum efforts?

RQ-4.1: How to satisfy requirements to work offline with minimum efforts?

RQ-4.2: How to configure the application with minimum efforts?

RQ-4.3: How to install the application?

6.1 Service workers

The key to work offline are service workers. They are JavaScript files where the developer uses cache API to store assets that need to be accessible offline. He also specifies how to handle requests: load a response from the cache or send a request to the internet using fetch API. It can also cache server responses. Both fetch API and cache API actively use the ES6 asynchronous programming concept called Promises that allows chain handlers instead of nesting them one into another. The Figure 7 shows an example of service worker where 4 assets are cached during its initialization that takes place during the first application execution. The “activate” event is fired when the service worker is updating. In this handler the developer can clear old caches. The “fetch” event is fired when the application makes a server request. In the Figure 7 the browser tries to find a response in the cache first, then it makes a request to the server if there is no cached response. When the server returns the response, then the browser caches it. Promise syntax can be quite tricky for unexperienced developers. In addition to its definition that must be in separate file, a service worker must be included in the HTML page and explicitly registered in JavaScript. If the developer simply wants to make some assets available offline then manually writing a service worker can result in too much additional work.

```
VERSION='v4';
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(VERSION).then(function(cache) {
      return cache.addAll([
        'icon192.png',
        'page.html',
        'script.html',
        'style.html'
      ]);
    })
  );
});
this.addEventListener('activate', function(event) {
  event.waitUntil(caches.keys().then(function(keyList) {
    return Promise.all(keyList.map(function(key) {
      if (key!=VERSION) {
        return caches.delete(key);
      }
    }));
  }));
});
this.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request).then(
    function(response) {
      return response || fetch(event.request).then(function(response) {
        return caches.open(VERSION).then(function(cache) {
          cache.put(event.request, response.clone());
          return response;
        });
      });
    })
  );
});
```

Figure 7. A service worker example

The main purpose of Service Workers is to allow the application to work offline by caching network responses. The main candidates for caching are static assets. The solution that is described in this thesis allows to define required assets like JavaScript libraries and CSS styles. This means that potential content of the cache is already known and does not require additional actions from the developer at all. Service workers and initialization code can be generated entirely. The only one question is what caching policy to use. The table below shows a list with possible caching policies.

Policy	**Description**
Cache only.	Searches only in the cache without using the network, good for static assets like scripts and stylesheets.

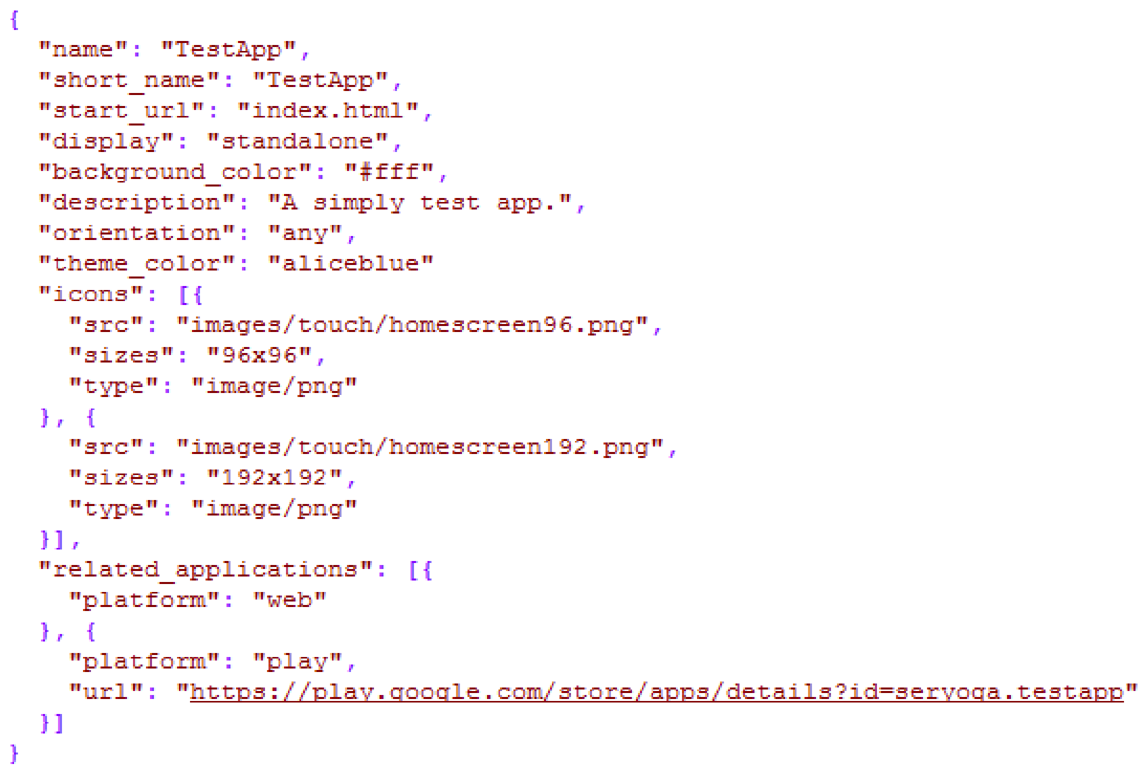
Network only.	Does not use the cache, good for dynamic data provided by the server.
Cache falling back to the network.	Searches first in the cache. If the result is found then returns it, else makes a network request.
Cache falling back to the network with cache update.	Like previous, but also caches the result of the sent network request.
Network falling back to the cache.	Makes a network request first. If there is no internet connection, then searches in the cache.

Table 7. Caching policies [21]

We want to cache all static assets, then the "network only" policy does not suit. "Cache only policy" also does not suit because it makes impossible to communicate with the server that makes impossible to develop online applications. Both "cache falling back to the network" and "network falling back to the cache" policies suit because they allow to provide responses online and offline. But we know what assets are static and must be retrieved from the cache, then the "cache falling back to the network" is more preferable.

6.2 Application manifest

The web application manifest is a simple JavaScript file in JSON (Java Script Object Notation) format. Unlike service workers that must be enabled in JavaScript, manifest must be enabled in HTML link tag. Example of a web application manifest is given in the Figure 8. It defines such properties like basic application appearance, application name and description, icons. In case of common web site similar items like page name, base URL (Universal Resource Locator) and icons are defined using HTML. But on the other hand, such parameters as display mode and background are unique for PWAs.



```
{
  "name": "TestApp",
  "short_name": "TestApp",
  "start_url": "index.html",
  "display": "standalone",
  "background_color": "#fff",
  "description": "A simply test app.",
  "orientation": "any",
  "theme_color": "aliceblue"
  "icons": [{
    "src": "images/touch/homescreen96.png",
    "sizes": "96x96",
    "type": "image/png"
  }, {
    "src": "images/touch/homescreen192.png",
    "sizes": "192x192",
    "type": "image/png"
  }],
  "related_applications": [{
    "platform": "web"
  }, {
    "platform": "play",
    "url": "https://play.google.com/store/apps/details?id=seryoga.testapp"
  }]
}
```

Figure 8. A web application manifest example

As was said before, the application manifest is just a configuration file in the JSON format. The configuration process can be simplified by using a GUI configuration form instead of writing code by hands. After that a proper manifest file can be generated.

To reduce number of actions, some fields can be generated entirely automatically. The table below contains a list of fields that the generated manifest file can contain. Display property defines what parts of system UI show to user. The standalone mode is the most preferable because in this case application looks like a native one (no browser UI, only system UI). Also a good application must work in any orientations, then there is no need to prohibit some orientations. The table below shows usable manifest fields with their creation mode.

Field	**Creation mode**
Name	Mandatory, Entered by user the using text field.
Short name	Same as the previous field.
Start URL	Generated automatically.

Description	Not mandatory, Entered by the user using a text field.
Display.	Generated automatically as standalone.
Background color	Selected using a colour picker, not mandatory.
Theme color	Selected using a colour picker, not mandatory.
Orientation	Generated automatically as any.
Icons	Selected using a file open dialog.

Table 8. Web application manifest fields

6.3 Installation

To be installable the PWA must satisfy some requirements. It has to have a proper manifest (has a name and a short name, has a 192px square icon, has a start URL), has to have a service worker, has to serve over HTTPS (Hypertext Transfer Protocol Secure) [22]. These requirements are controlled by the browser. If they are satisfied, then it shows an installation banner (prompts the user to install the application).

2 of 3 specified requirements are satisfied by solutions developed before. The only one requirement left is the HTTPS requirement. This means that to be installed the application must be served using a web server. After it was installed, there is no need to serve this application anymore. The entire installation process can be described using the following steps:

1. After clicking the “install” button the building tool sends the generated content to the server
2. The web server stores it and returns an URL where it can be accessed
3. The web server periodically cleans applications that was not accessed during specified period of time. Identification can be done by the name of the application.

6.4 Evaluation

The table below contains the analytical evaluation of the developed conceptual solution for the PWA producing process. The result column contains pros of the solution. It can also contain cons. In this case the optimal conditions are given.

Solution	**Result**
Generate service workers entirely automatically using predefined cache policy.	Pros: No need to deal with service workers manually anymore that excludes coding errors and speeds up developing process. Cons: Only one caching policy. Optimal conditions: All caching assets are known at the building time. No requirement to cache something dynamically during execution time.
Configure the application using a GUI input form and generate its web application manifest using the entered data .	Pros: Configuration in the intuitive way. No need to deal with code that excludes errors and speeds up development. Colour selecting instead of writing raw hexadecimal code. Cons: Some manifest fields like related applications cannot be edited. Optimal conditions: Basic configuration requirements.
Possibility to install the application.	Pros: The developer gets his fully working application “in one click”. The application exists entirely separately from the ecosystem developed in this thesis.

Table 9. PWA producing process evaluation

7 Implementation of the component creator

As was shown in previous chapters, the conceptual solution for the problem of this thesis consists of 4 parts that are reflected in the research questions:

1. Component creation
2. Final application building
3. Components management
4. PWA transformation

PWA transformation can be treated as a final step of application building. Components management connects processes of application building and component creation. Another words, the implemented solution can consist of these 2 parts:

1. An environment for component creation
2. An environment for final application building.

This chapter is dedicated to component creator implementation. It covers the following parts of the conceptual solution:

- Component creation
- Components management

Implementation requirements will be formulated based on the conceptual solution described in the previous chapters. Based on these requirements design of the application will be developed in the form of UML diagrams using IBM Rational Rose CASE (Computer-Aided Software Engineering) tool. After that developed design can be implemented using programming tools. Vue.js is a web frontend framework and the easiest way to integrate it with the development environment is to use the same environment where Vue.js works. Another words the solution must be implemented in the form of a web application. This decision allows the solution to run on multiple platforms. And to make it connection-independent and installable, it can be implemented in the form of a PWA by itself.

7.1 Requirements specification

Here is shown a list of requirements that are specified based on the conceptual solution evaluation part of the chapters related to component creation and components management. A single user for all these requirements is the developer.

Component creation:
When the user finishes with code editing, changed parts are processed depending on their type. Resulting code is pasted into component's wireframe (or code template) of the preview page. After that the resulting page is shown using a preview iframe.
The user can watch generated code of the component instead of the component preview.
If there an exception occurs, then display its information in the preview iframe such as used Vue.js version, the type of exception and its text, source of exception (a line number with the nearest code lines).
The user can open a template editing field by clicking an appropriate button. The template editing field has a HTML syntax highlight. When the user closes this field, slot definition entries are searched in edited code and a list of slots is updated. After that the preview must be updated.
The user can open a properties editing field by clicking an appropriate button. Properties have syntax "[type] name [default value]" where brackets mean optional parts. Properties have own highlight mode. When the user closes this field properties are transformed into Vue.js syntax and the preview must be updated.
The user can open a style editing field by clicking an appropriate button. The style editing field has a CSS syntax highlight. When the user closes this field the preview must be updated.
The user has a list of fixed hooks (created, mounted, updated, destroyed). He can open a code editing field by clicking one of them.
The user has editable lists of methods and computed properties. He can add a new method or computed property by clicking the "add" button. After that he can enter a name of the created method or computed property. When he has done this, a code editing field is opened immediately and a generated empty function is created. He can delete the previously created method or computed property by clicking the "delete" button next to its name in the list.
Code editing fields that are used for hooks, methods, watchers and computed properties has JavaScript syntax highlight. When these fields are closed, the following processing is applied to them: event emitting entries are searched in code, data initialization entries are searched in code. After that watchers and events are updated followed by the preview update.

Found data initialization entries are used to generate the data field. These entries are displayed in a list of watchers. The user can edit watcher by clicking its name in the list. This opens the previously described code editing field.
Show a lists of defined events and slots to the user
Possibility to collapse lists. Group template, properties, style and include into a separate list (unlike other editable items that have JavaScript code type, they have other code type).
Components management:
By clicking the “download’ button the user can download generated code of the created component in a specified format. Generation works similar with the preview code generation. The difference is that another code wireframe is used.
By clicking the “save” button the user can save his component into the component library. The component creator will ask for the component’s name. If it was opened, then the name field will contain its name that can be edited. Components are identified by their names and providing the name of the existing component will result in its overwriting.
By clicking the “open” button the user can open a list of components stored in the component library. He can load a needed component by selecting it from this list.
Provide an input field where the user can specify outer assets that his component uses. When the user closes this field appropriate HTML tags are generated to include these assets.
Other:
Implement the solution in the form of a PWA by providing a service worker, an icon and an application manifest file
Add a configuration dialog where the user can select API and library URLs; configure preview.

Table 10. Component creator requirements

7.2 Design

The overall GUI of the component creator must be simple, without complex details. It must contain a preview iframe, a lists of component parts like methods and hooks with the collapse possibility, a toolbar with buttons for component management. The figure below shows the overall GUI of the component creator.

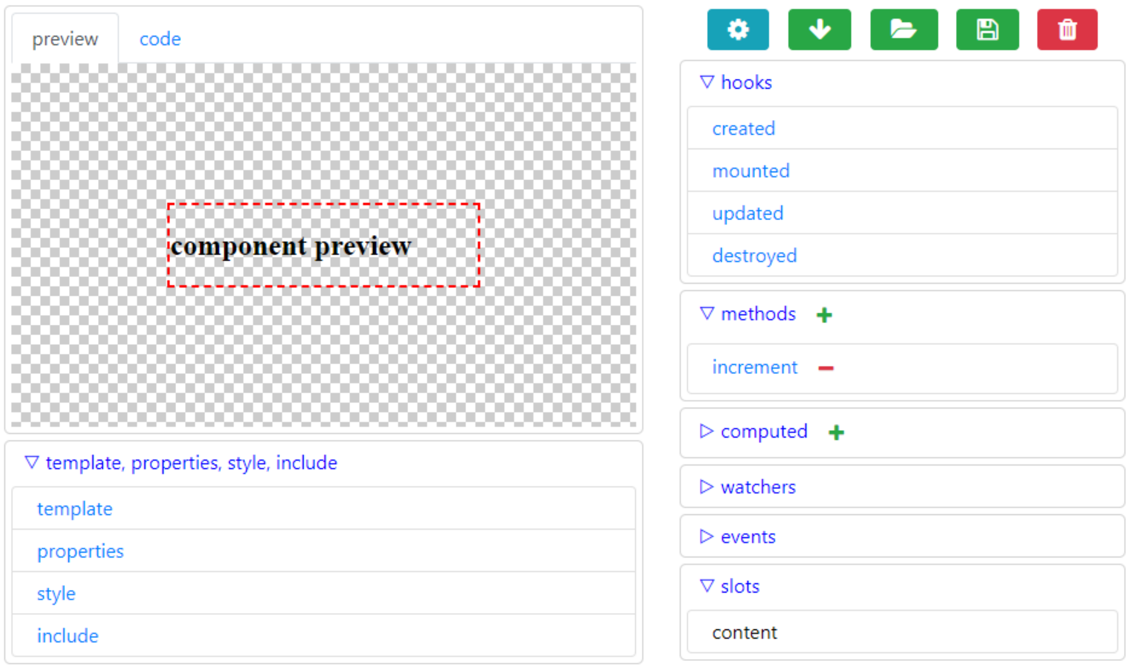


Figure 9. The GUI of the component creator

The component creator consists of 3 parts according to the MVP pattern: The Model part is independent from other parts and provides such functionality as required component processing, interaction with component library and code generation. The Presenter part handles input events and calls methods of the Model part or uses its properties. It also uses the View part to render the Model. The View part is presented with the DOM and not displayed on the figure below. The Figure 10 shows main parts of the component creator that will be implemented using OOP classes. CRUD means here Create Read Update and Delete operations.

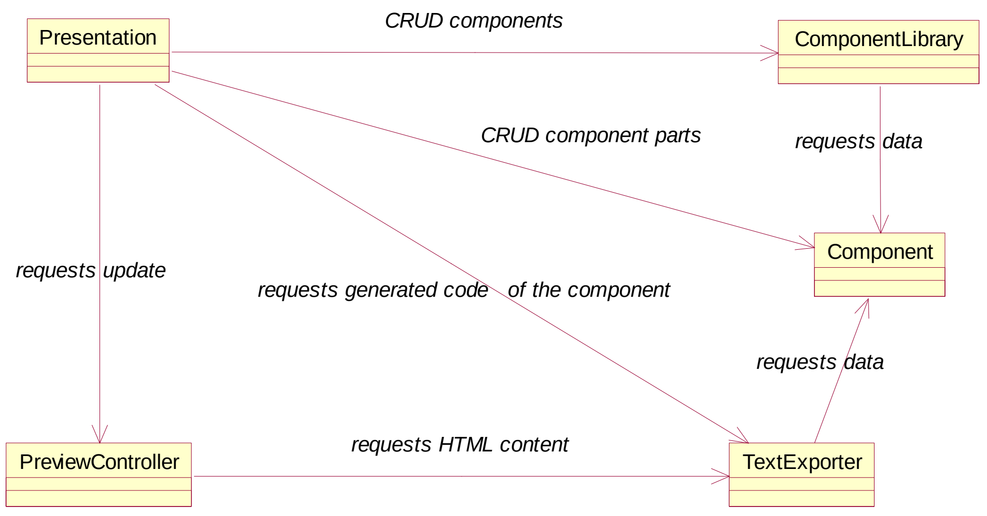


Figure 10. The main parts of the component creator

Component has function-type fields that need special processing like computed properties, watchers, hooks and methods. This requires to make separate collection-like class to work with them. There are 2 types of component libraries: the local one and the remote one. This requires to make 2 separate classes for each of them. Also there is need to generate different code for preview and for download. This requires 2 separate classes for text exporting. The figure below shows resulting class diagram with fields, methods and properties (contain names only). Also quantitative relationship is shown.

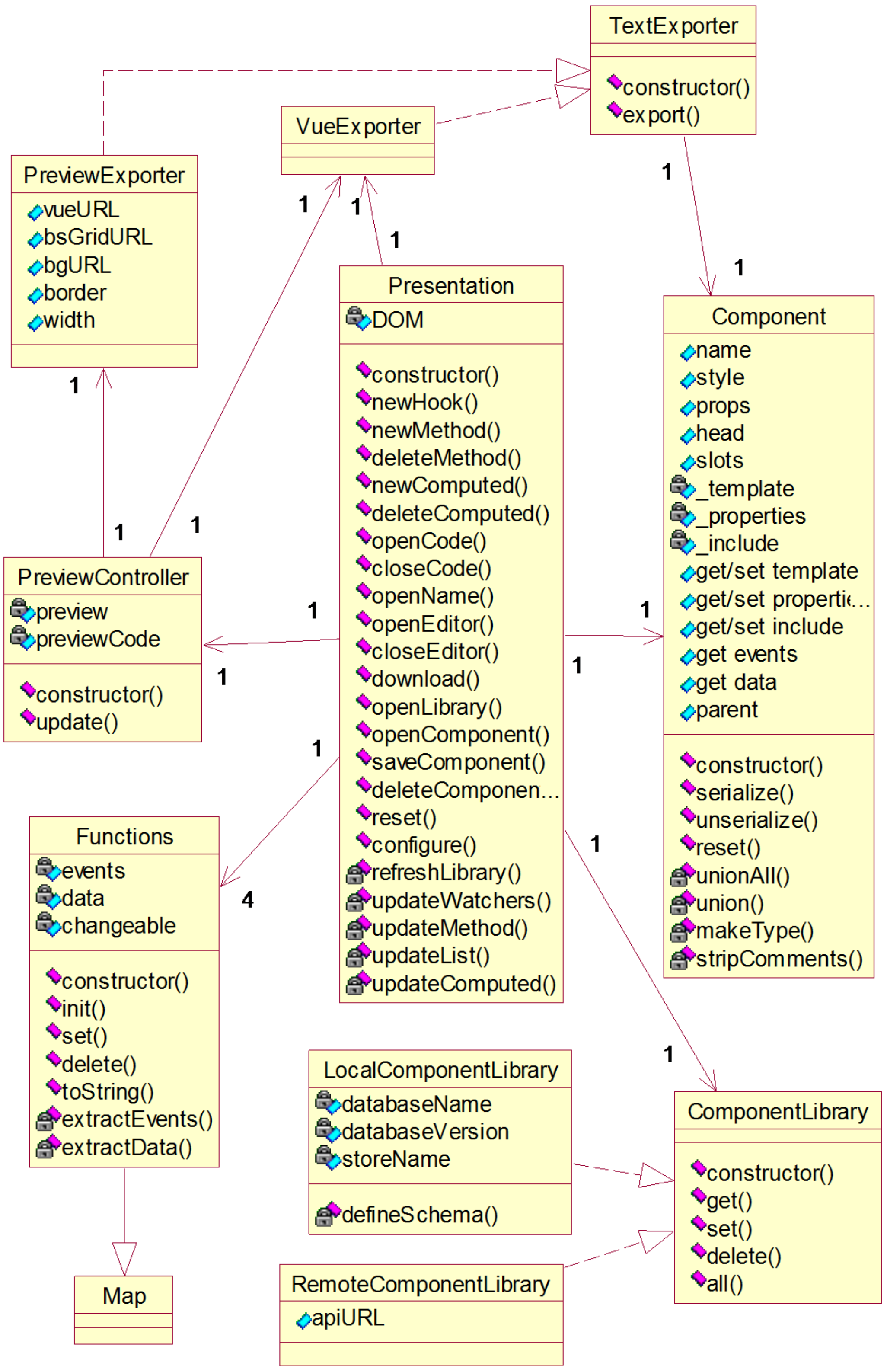


Figure 11. The component creator class diagram

The following table contains detailed description of the class members that was shown in the figure above. Member definitions uses possibilities that ES6 has like properties, introspection, Maps and Sets. To make picture more clear these definitions are written with their expected type specified (ES is a dynamically typed language). This table has signatures of class fields in the left column and a description in the right column.

Presentation	
constructor()	Initializes the View and the Model by calling their constructors. Registers event handlers for View events. Creates a list of predefined hooks (created, mounted, updated, destroyed) using newHook(). Initializes the configuration dialog by calling configure().
View DOM	Represents all View elements.
void newHook(string name)	Creates a new hooks with passed name for the component and updates a list of hooks in the View.
void newMethod(string name)	Creates a new method with the passed name for the component and updates the View using updateMethods(). Opens the code editor for created method using openCode().
void deleteMethod(string name)	Deletes a component method by the passed name using Functions.delete() and updates the View using updateMethods(), updateWatchers(), updateList().
void newComputed(string name)	Creates a new computed property with the passed name for the component and updates the View using updateComputed(). Opens a code editor for created computed property using openCode().
void deleteComputed(string name)	Deletes a component computed property by the passed name using Functions.delete() and updates the View using updateComputed(), updateWatchers(), updateList().
void openCode(string type, string name)	Opens the function code editor (for function-type fields like watchers, methods, computed and hooks) and loads code specified by type and name from the component. Sets a handler for the close editor event which uses closeCode().

void closeCode(string type, string name)	Saves code specified by the passed type and name and edited in the function code editor into the component. Updates the View using updateList() and updateWatchers(). Updates the preview using PreviewController.
void openName(View inputField, function callback)	Opens the name input field and sets the specified function as a close editing field event handler.
void openEditor(View editor, string type)	Opens the code editor (for non-function-type like template, style, properties and include) specified by the first argument and loads code specified by type from the component.
void closeEditor(string type)	Saves code specified by type into the component and edited in the code editor. Updates the View using updateList() for slots in case of template. Updates a preview using PreviewController.
void updateList(View list, Functions items)	Updates the list (View part) specified by the passed list using the passed items object.
void updateMethods()	Updates the list of methods (View part).
void updateComputed()	Updates the list of computed (View part).
void updateWatchers()	Updates the list of watcher (View part). Synchronizes watchers with the data: creates watchers for new data fields and deletes watchers for deleted data fields.
void download()	Converts the component into the .vue format using VueExporter and downloads the resulting file.
void openLibrary()	Loads a list of components using ComponentLibrary and calls refreshLibrary().
void openComponent(string name, boolean isRemote)	Loads the component specified by the name using ComponentLibrary into the creator, updates the View using updateMethod(), updateComputed(), updateWatchers(), updateList() and updates a preview using PreviewController.
void saveComponent(string name)	Saves the currently opened component with the specified name using ComponentLibrary.
void deleteComponent(string name, boolean isRemote)	Deletes the component specified by the name using appropriate ComponentLibrary depending on the second argument then calls refreshLibrary().

void refreshLibrary(string[] names, boolean isRemote)	Refreshed library View using passed component names and isRemote flag.
void reset()	Resets state of the current component; updates the View using updateMethod(), updateComputed(), updateWatchers(); updates a preview using PreviewController.
void configure(boolean configure)	Uses the configuration View to change some parameters of the preview and the component library like Bootstrap grid URL and remote component library API URL. Updates the preview using PreviewController. If configure=true, then applies changes, else loads configuration data into the configuration View.
Component	
constructor()	Resets its state using reset().
string name	The name of the component.
string parent	The name of the component that was used for forking.
string style	Contains entered CSS style.
string props	Contains the generated Vue.js props field.
string head	Contains the generated html for external assets inclusion.
Set<string> slots	The set of extracted slots.
string _template	Contains entered template definition.
string _properties	Contains entered properties definition.
string _include	Contains entered external assets definition.
string property template	Get: returns _template. Set: assigns a new value to _template, extracts slots.
string property properties	Get: returns _properties. Set: assigns a new value to _properties. Generates Vue.js properties and stores them in the props using makeType() and stripComments().
string property include	Get: returns _include.

	Set: assigns a new value to _include. Generates HTML asset inclusion tags and stores them into head using stripComments().
readonly Set<string> property events	Get: returns united event names using unionAll() for events.
readonly string property data	Get: returns the generated Vue.js data field using unionAll() for data.
Set union(Set set1, Set set2)	Adds set2 to set1 and returns set1.
Set<string> unionAll(string dataType)	Returns united data of all Functions instances (watchers, methods, computed, hooks) specified by datatype argument using union().
string makeType(string word)	Returns a type-like word by casting passed word to lowercase and capitalizing a first letter.
string[] stripComments(string lines[])	Removes comments from passed strings and returns the resulting array also removing empty strings.
Object serialize()	Returns an object that contains minimal number of fields to restore this component using unserialize().
void unserialize(Object image)	Resets the state with reset() and restores component state using passed object previously created with serialize().
void reset()	Resets to the initial state clearing all data.
Functions (extends built-in Map<string,string>)	
constructor(boolean changeable)	Initializes fields.
void init(string name, string …args)	Generates empty function code using the passed name and arguments then saves it.
void set(string name, string code)	Updates passed function code identified by name. Extracts and stores data fields and events using extractData() and extractEvents(). If changeable, the function name is extracted from code and used as a key. Else function name cannot be changed and will be equalized with the key value.
void delete(string key)	Removes a function using the passed key. Also removes it from events and data.

String toString()	Returns generated code of the function-type Vue.js component field based on functions stored in this Functions instance.
Set<string> extractData(string code)	Extracts and returns data fields from passed code.
Set<string> extractEvents(string code)	Extracts and returns events from passed code.
Map<string,Set> events	Contains a set of events extracted from each function.
Map<string,Set> data	Contains a set of data fields extracted from each function.
boolean changeable	Are functions stored in this object can change their names.
VueExporter (TextExporter implementation)	
constructor(Component source)	Stores the passed component to use it as a data source for code generation.
string export()	Returns generated code of the component in the .vue file format.
PreviewExporter (TextExporter implementation)	
constructor(Component source)	Stores the passed component to use it as a data source for code generation.
string export()	Returns generated page that has registration of the component and generated preview HTML layout based on the configuration. The generated page also posts error message to the parent window.
string vueURL	The URL to download Vue.js framework.
string bsGridURL	The URL to download Bootstrap Grid.
string bgURL	The URL of the preview background.
string border	Component preview bounding border style.
int width	Component preview bounds width (in Bootstrap Grid units).
PreviewController	

constructor(View preview, PreviewExporter previewSource, View codePreview, VueExporter codeSource)	Stores passed code exporters to use them as code source in future. Stores passed Views to use them for result displaying. Registers error handler to display errors in the passed preview iframe.
void update()	Updates the content of the preview iframe using code provided by PreviewExporter or content of code preview using VueExporter.
View preview	The preview iframe to display a preview result.
View previewCode	The read-only code area to display generated code.
LocalComponentLibrary (ComponentLibrary implementation)	
constructor()	Initializes fields.
Object get(string name)	Returns a serialized representation of the component specified by the passed name from the local component database.
void set(Object component)	Inserts a component into the local component database using its serialized representation.
void delete(string name)	Deletes a component from the local component database specified by passed name.
string[] all()	Return all component names that are stored in the local component database.
void defineSchema(Event event)	Defines a schema of the used local component database.
const string databaseName	The local database name.
const string databaseVersion	The local database version.
const string storeName	The local store name.
RemoteComponentLibrary (ComponentLibrary implementation)	
constructor()	Initializes fields.
Object get(string name)	Returns a serialized representation of the component specified by the passed name using the remote component library API.
void set(Object serialized)	Inserts a specified serialized component using the remote component library API.

void delete(string name)	Deletes a component by name using the remote component library API.
string[] all()	Returns all component names that are stored in the remote component library using its API.
string apiURL	The API URL that serves the remote component library

Table 11. Description of component creator classes

7.3 Implementation

The following section contains remarks and nuances of the implementation process. The component creator prototype was made in the form of a web application. It uses JavaScript (ES6) as its programming language, HTML and CSS for the View definition. To interact with the View from the Presenter jQuery library was used (http://jquery.com/). To assist with the requirement of responsive design Bootstrap framework was used.

As a code editor was used CodeMirror (https://codemirror.net/). It has modes with syntax highlight for HTML, CSS and JavaScript. Its mode/simple add-on allows custom modes to be specified using a relatively simple declarative format. This format is not as powerful as writing code directly against the mode interface, but is a lot easier to get started with, and sufficiently expressive for many simple language modes. This mode uses a concept of a state machine. Each state has a regular expression that match a part of code and a token that defines what to do with this part. If the regular expression matches, then specified token is applied to matched part and the next state can be selected [23]. The figure below shows a state diagram that represents a syntax highlight for custom property syntax. The processing is ended when the end of string is reached.

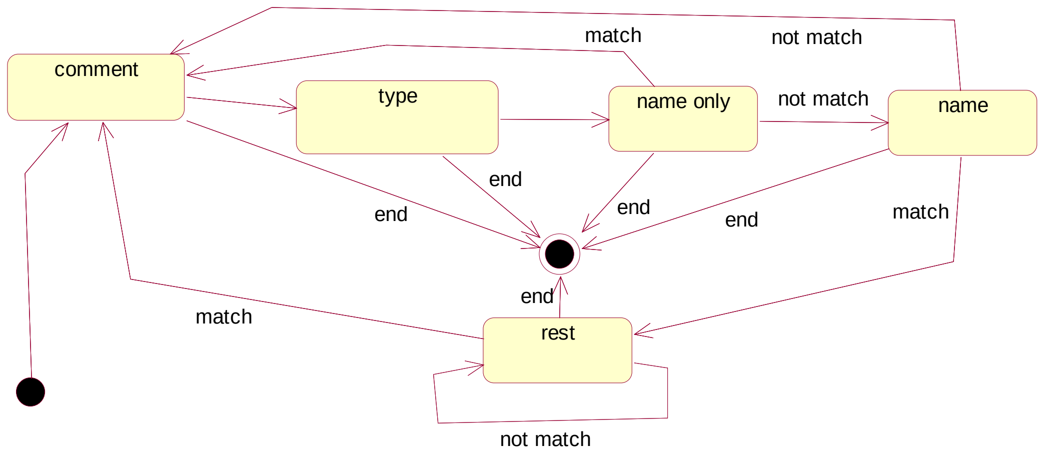


Figure 12. The custom property syntax highlighting state machine

Regular expressions were also used for code processing to extract data for example. The table below contains a list of used regular expressions (in the JavaScript format) with a use case description. They were used alongside with common imperative programming where their application was not enough.

Regular expression	**Description**
/(string|number|boolean|function|object|array|symbol)(\s+)/i	Highlight: type
/#.*$/	Highlight: comment
/[a-z0-9_$]+(?=[=]*$)/i	Highlight: property name only
/[a-z0-9_$]+(?=[=]*)/i	Highlight: property name
/[^#]*/i	Highlight: rest of string (expects value)
/<slot.*name=['"]\w+['"]/ig /name=['"]\w+['"]/i	Extract: a slot definition and then a slot name
/\S+/g	Split a custom property string
/\.css$/i /\.js$/i	Get format of assets in include field by extension
/this\.[a-zA-Z0-9_$]+[=\s;}]/g	Extract: data
/this\.\$emit\(['"]\w+['"]/g	Extract: events

/[\$_a-z0-9]*(?= \s* \()/i	Extract: function name

Table 12. Used regular expressions (component creator)

As a local storage for component library was used IndexedDB. It was chosen, because it allows to store JavaScript objects (called documents), use indexes and has quite huge limit. Other candidates are: LocalStorage, ApplicationCache and WebSQL. It was chosen over WebSQL because the second technology is considered as its predecessor. The Table 14 shows limits of these technologies [24]. IndexedDB is a transactional database system like SQL-based RDBMSs (Relational Data Base Management Systems). However, unlike SQL-based RDBMSes, which use fixed-column tables, IndexedDB is a JavaScript-based object-oriented database. IndexedDB lets you store and retrieve objects that are indexed with a key. Any objects supported by the structured clone algorithm can be stored. You need to specify the database schema, open a connection to your database, and then retrieve and update data within a series of transactions [25]. IndexedDB uses an asynchronous approach via callbacks. To make it synchronous these callbacks can be wrapped into Promises and used with the ES6 asynchronous programming possibilities (like async and await). The table below shows the schema (structure of the documents) that was used for the local component library.

Field	**Description**
Name (indexed)	Component’s name (used as a key)
component	Stores a serialized component representation

Table 13. The local component library IndexedDB schema

Technology	**Chrome (40+)**	**Firefox(34+)**	**Safari(8)**	**IE(10,11)**
IndexedDB	100+MB (quota)	100+MB	100+MB	100+MB
LocalStorage	10MB	10MB	5MB	10MB
WebSQL	100+MB (quota)	100+MB	100+MB	Not supported
ApplicationCache	100+MB (quota)	100+MB	100+MB	100+MB

Table 14. Browser storage limits [24]

The Table 11 displays general structure of the component creator. During the implementation phase there was also added additional inline callbacks (that also locate in HTML). JavaScript does not allow type definition, but this table shows what type of data is expected to store in the specified variables. The View type represents a part of the MVP pattern's View. Views are DOM nodes that are queried using jQuery. jQuery queries using CSS selector and in the implementation Views are passed by CSS selectors (strings). In case of web applications View part is defined using HTML and styled using CSS. Editors are represented by CodeMirror instances.

Asynchronous IndexedDB callbacks was made synchronous using ES6 Promises and the await keyword. An asynchronous function wraps its functionality into the Promise instance and immediately returns it while continues execution in the separate thread. The Promise will be fulfilled when asynchronous code will call its callback with the execution result. The await keyword wait stops execution until its Promise will be fulfilled.

For the preview an iframe was used. Iframe represents a child browser window that is located inside parent's page. PreviewExporter class generates required HTML page with additional error handlers. These handlers send message to the parent window (where the component creator works). The component creator intercepts them and displays errors generating another HTML page that is displayed in the same iframe. HTML pages are set using data: URL that allow to define content directly in the URL. To overcame browser caching a random number was generated as a comment string in the beginning of the HTML page.

There is no Presenter class in the implementation. JavaScript does not require to put all code inside classes. Presenter was implemented using a set of functions and variables that locate in the global scope (as fields of the browser's window object instance).

The standard workflow of the component code generation can be illustrated using these steps (from making changes to auto preview update):

1. The user clicks required field, for example created hook. Browser calls an inline HTML callback that calls openCode("hook", "created"). (for non-code fields openEditor() is called).

2. openCode() loads field content from the component instance using get() method of the hooks property, readies the “onclose” callback where it calls closeCode() with appropriate arguments and shows code editing field in the modal window. (closeEditor() is used by openEditor())

3. When the user has finished with code editing and has closed the editing modal, the “onclose” event is fired and closeCode(“hook”,”created”) is called. (closeEditor() uses an appropriate getter of the component).

4. closeCode() saves the edited code to the component instance using set() method of the hooks property. (closeEditor() uses an appropriate setter of the component).

5. The hooks property is Functions instance. Its set() method uses extracting method like extractEvents() and extractData() for the entered code processing. (Component setters that are called by closeEditor() have appropriate processing that suit for their type).

6. closeCode() calls updateList() to update the list of events, updateWatcher(), updateMethods(), updateComputed(). This synchronizes the View with the previously made changes. (closeEditor() calls only updateList() to update the list of slots).

7. closeCode() uses update() method of the PreviewController instance. (closeEditor calls it also).

8. update() method of the PreviewController uses appropriate TextExporter (PreviewExporter or VueExporter) depending on the currently selected preview mode and calls its export() method to get generated code and update the iframe content.

9. export() method of the appropriate TextExporter uses the Component instance to retrieve required data. Component by itself deals with code generation and text exporter assembles the resulting code using the code provided by the component and skeleton code that depends on the exporter type. This is done using ES6 string template syntax.

8 Implementation of the application builder

This chapter is dedicated to the application builder implementation. It covers the following parts of the conceptual solution:

- Application building
- Components management
- PWA transformation

Implementation requirements will be formulated based on the conceptual solution described in the previous chapters. Based on these requirements design of the application will be developed in forms of UML diagrams using IBM Rational Rose CASE tool. After that developed design can be implemented using programming tools. Application builder will be developed in the form of a web application as previously developed application creator.

8.1 Requirements specification

Here is shown a list of requirements that are specified based on the conceptual solution evaluation part of chapters related to the final application building, component management and PWA producing. The single user for all these requirements is a developer.

Application building:
The user builds his application layout using such concepts as layers, rows and columns (can be component or row) in the automatically updating layout preview. The layout preview displays correct width, offset, align and paddings for columns. Columns are displayed using fixed height. Components display their names. Layers display their position (fixed-top, fixed-bottom, fixed-center, absolute, static).
The user can add a new layer by clicking the “add layer” button Then he can select this layer in the layout preview to edit its properties such as position. He can delete the selected layer by clicking the “delete” button.
The user can add a new row by clicking the “add row button” when a row or a layer is selected. Then he can edit row layout properties such as width, offset, align, padding. He can delete the selected layer by clicking the “delete” button.

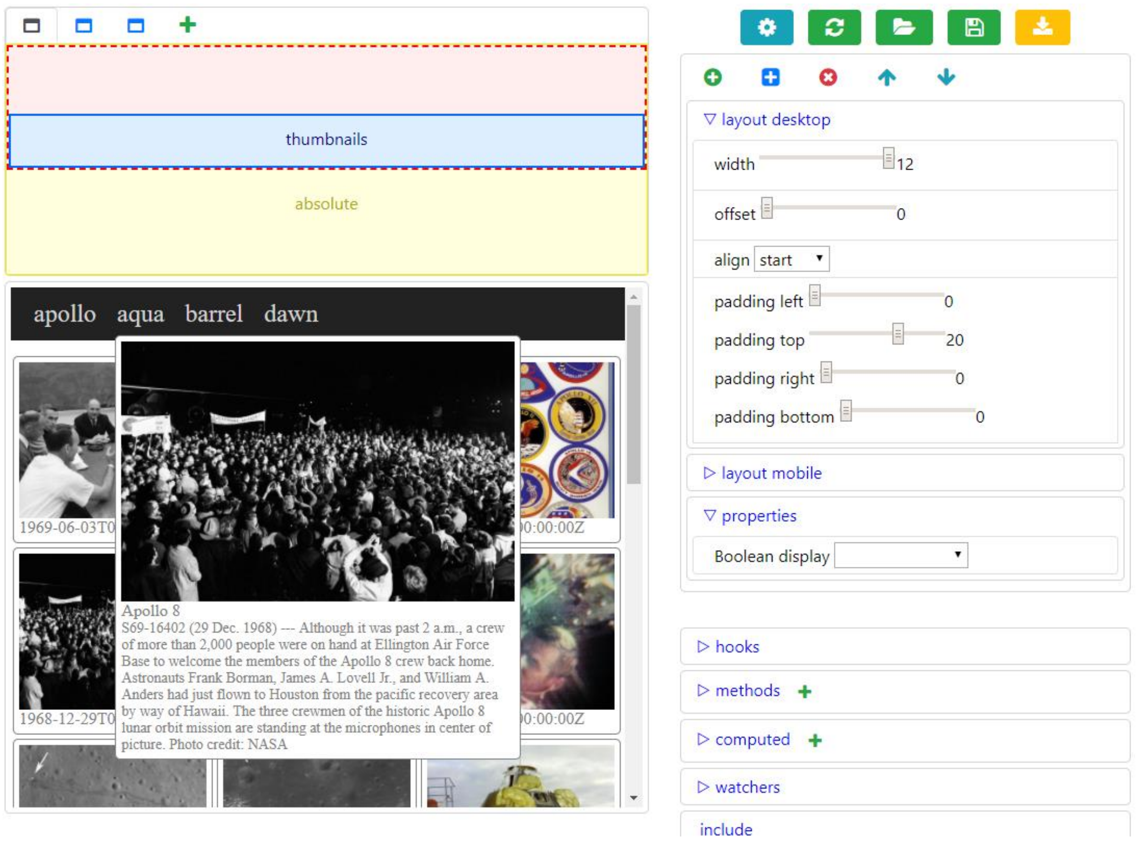
The user can add a new component by clicking the “add component button” when a row is selected. Then he can edit component layout properties that are similar with row layout properties. He can delete the selected layer by clicking the “delete” button.
The user can change the order of children inside selected element by clicking the “up” and “down” buttons.
The user can edit 2 separate sets of layout properties: one for mobile and another for desktop.
All elements (layers, rows, components) have reactive display property that uses conditional rendering of Vue.js.
The user can click the “style” button and edit component style in a code editing field with CSS syntax highlight.
The user can select a required component slot from a list and edit it in a code editing field with HTML syntax highlight.
Allow user to bind his ViewModel with the component ViewModel by selecting property values from a dropdown list (contains application ViewModel variables). Show property type if it was defined in the component.
Provide the user with a list of predefined empty event handlers for all events that component emits. The user can select required handler from the list and edit its code in a code editing field with JavaScript syntax highlight.
Provide the user with a lists of all component methods (with arguments) and computed properties.
Provide the user with the functionality analogical with component creator to work with the application ViewModel. Unlike component creator, it contains only hooks, methods, computed properties and watchers lists and appropriate JavaScript editing fields.
The user can press “refresh” button that will trigger the application building process. When the application will be built, the user can test it in the preview window that has 2 views: desktop and mobile.
When the user adds a new component, he can specify a component name. He can reference the component from the code using this name as an application ViewModel variable.
Analyse component code to find scoped slots and use their attributes to prepend variables used inside the slot.
Analyse component code to find scoped property modification and replace with event emitting (provides two-way data binding).
If there an exception occurs, then display exception information in the preview window (analogically with the component creator).

Component management
When the user clicks the “add component” button, the component library is shown and the user can select the required component from there. After that component is loaded into the application builder.
Provide a field where the user can specify his outer scripts and assets like CSS (analogically with the component creator).
Transformation to PWA
The user can open the configuration window where he can specify manifest fields. Appropriate input methods must be used for different fields as was shown in the Table 8**.**
By clicking the “install” button application will be sent to server where the manifest and service worker will be added to it. After that the returned link will be used to navigate to the application.
Other
Implement the solution in the form of a PWA by providing a service worker, icon and an application manifest file.
Extend the configuration dialog where the user can also select API and library URLs.
By clicking the “save” button the user can save his application giving it a name for future access or sharing with other people.
By clicking the “open” button the user can open previously saved application to continue editing. This shows a list of all applications where the user can select the required one.

Table 15. Application builder requirements

8.2 Design

The application builder has similar GUI requirements as the component creator. The figure below shows overall GUI of the application builder.



width 12
offset 0
align start
padding left 0
padding top 20
padding right 0
padding bottom 0
Boolean display

Apollo 8
S69-16402 (29 Dec. 1968) --- Although it was past 2 a.m., a crew of more than 2,000 people were on hand at Ellington Air Force Base to welcome the members of the Apollo 8 crew back home. Astronauts Frank Borman, James A. Lovell Jr., and William A. Anders had just flown to Houston from the pacific recovery area by way of Hawaii. The three crewmen of the historic Apollo 8 lunar orbit mission are standing at the microphones in center of picture. Photo credit: NASA

Figure 13. The GUI of the application builder

The application builder has MVP design pattern analogically with the component creator. The figure below shows the main parts of the application builder that will be implemented using OOP classes.

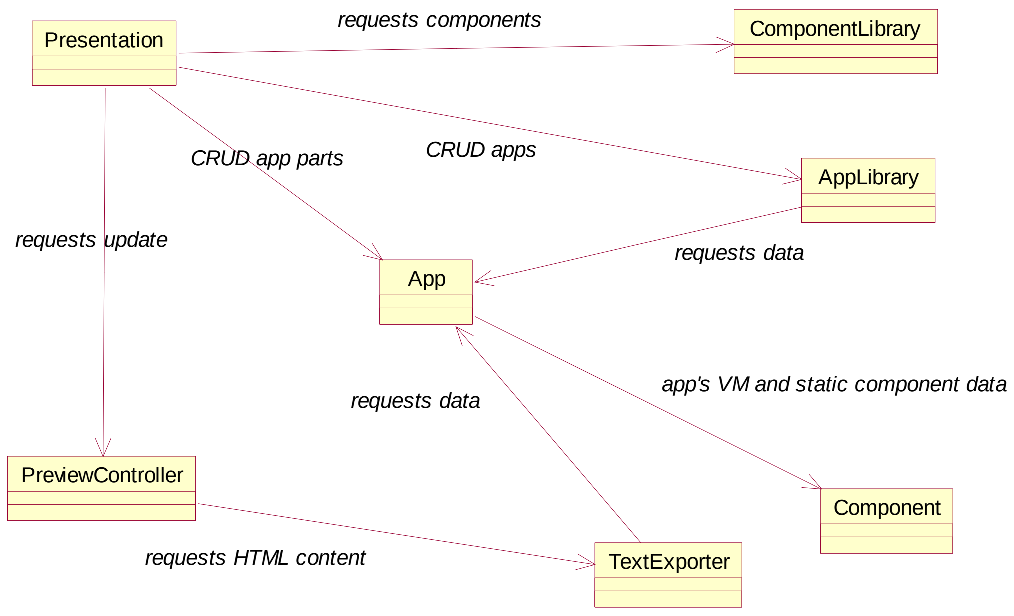


Figure 14. The main parts of the application builder

The App part from the previous figure consists of layers, layers consist of rows and rows consist of columns. A row can be nested to another row. This means that it must be a column itself. Leaf columns of this hierarchy contain components. This can be designed using the Composite GoF (Gang of Four) design pattern that describes a group of objects that is treated the same way as a single instance of the same type of object. [26]. The application library like the component library can be implemented locally (using one of the browser storage API) and remotely (using web API and server implementation). Moreover, it shares analogical structure with the component library classes and can extend them. Also some methods from the component creator's presentation can be used in the application builder. The solution requires additional CSS styles that must be used by both: the application builder and a building application. This requires additional text exporter that generates CSS code. The figure below shows a resulting class diagram with fields, methods and properties (contain names only). Also quantitative relationship is shown. Some classes were defined in the component creator and can be reused in the component builder. These classes are not described here (but are shown on the figure below without fields).

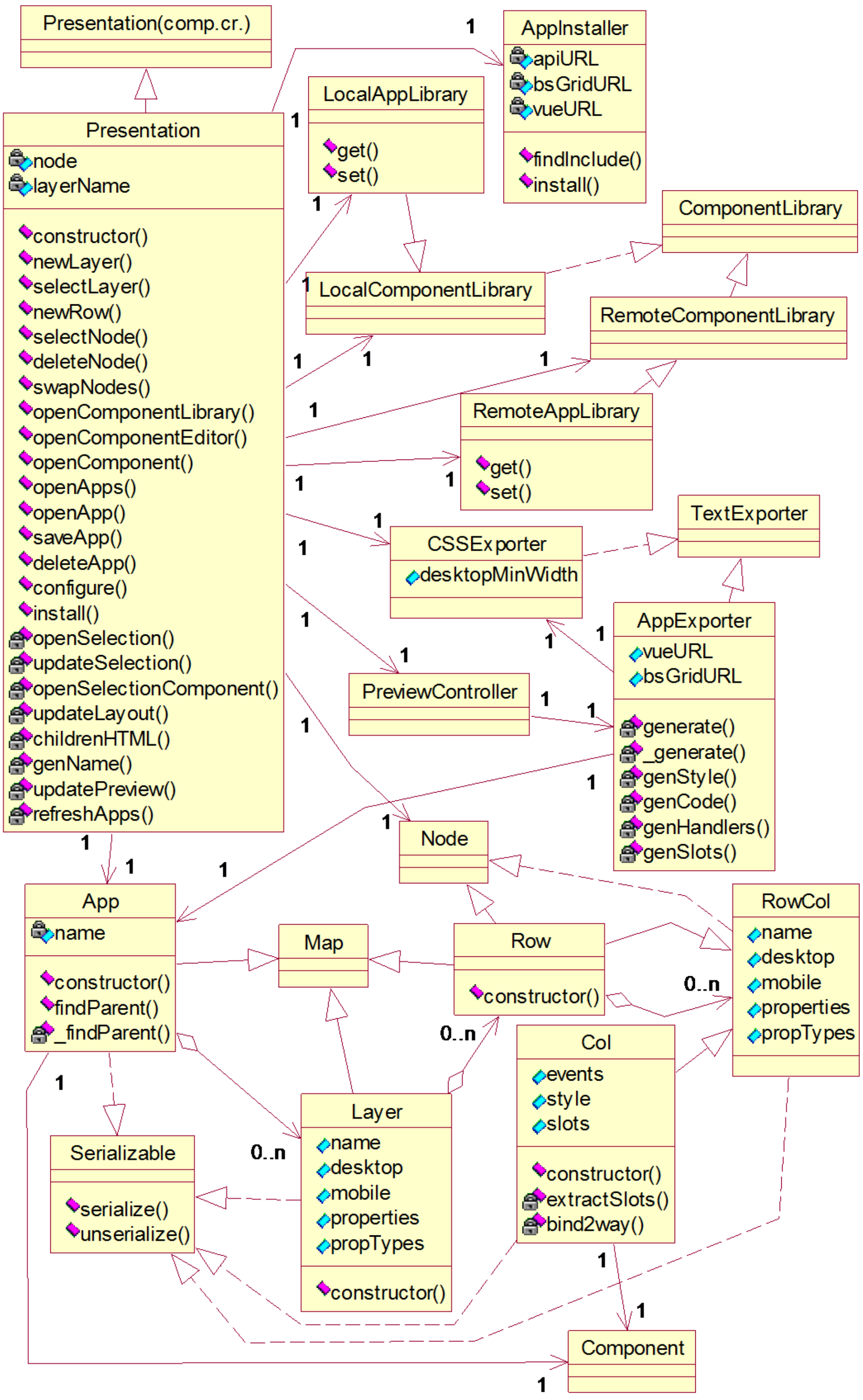


Figure 15. The application builder class diagram

The following table contains detailed description of the class members that was shown in the figure above. Member definitions uses possibilities that ES6 has like properties, introspection, Maps and Sets. To make picture more clear these definitions are written with their type specified. This table has a signature of the class fields in the left column and a description in the right column.

Presentation (implements Serializable)	
constructor()	Initializes the View and the Model by calling their constructors. Registers event handlers for View events. Creates a list of predefined hooks for application ViewModel (created, mounted, updated, destroyed). Generates required styles using CSS Exporter. Calls configure().
string layerName	The currently selected layer name.
void newLayer()	Creates a new Layer with a generated name using genName(), adds it to the application tree and selects it using selectLayer().
void selectLayer(string name)	Sets layerName, and calls openSelection(), updateLayout().
void newRow()	Creates a new Row with a generated name using genName(), adds it to the currently selected node and selects it using selectNode().
void selectNode(string name)	Calls openSelection() and updateLayout().
void deleteNode()	Removes currently selected node from its parent and calls selectNode() or selectLayer() depending on its type.
void swapNodes(boolean inv)	Swaps with the next node or the previous node (if inv), calls updateLayout().
void openSelection(string name)	Sets currently selected node using the passed name. Shows or hides builder parts depending on the type of the selected node and synchronizes it with the node state. Calls openSelectionComponent() if current node is a Col instance.

void updateSelection(boolean isLayer)	Synchronizes the currently selected node state with the View, calls updateLayout(). Uses another View parts in case of layer.
void openSelectionComponent()	Synchronizes the View with the currently selected node state (component specific).
void openComponentEditor(View modal, string type, string name)	Opens specified code editing modal and loads specified data. Sets a handler that updates the data after the modal is closed. (like openCode(), but for component used in the builder)
void updateLayout()	Refreshes the layout preview based on the currently selected component and layer using childrenHTML().
string childrenHTML(Node parent,Node, boolean l1)	Returns a HTML code for layout preview for single node. Root call has l1=true. If the node has children then continues recursion.
string genName()	Generates and returns an unique name.
void openComponentSelection()	Shows the list of available components using the LocalComponentLibrary and RemoteComponentLibrary.
void openComponent(string name, boolean isRemote)	Loads a component by the passed name using appropriate component library depending on the last argument. Then creates a new Col based on the loaded component and selects it using selectNode().
void updatePreview()	Uses PreviewController to update the preview.
void openApps()	Opens app library window and refreshes it using refreshApps().
void openApp(string name)	Loads specified app by name using app library. Then updates the builder by calling updateMethods(), updateComputed(), updateWatchers(), selectLayer(), selectNode().
void saveApp()	Saves the app using the app library.
void deleteApp(string name)	Deletes the app by name using the app library and calls refreshApps().
void refreshApps(string names[])	Updates the app library window using the passed app names.

void configure(boolean configure)	Uses configuration View to change some parameters of the application builder. If configure=true, then applies changes, else loads them into the configuration View.
void install()	Triggers the installation procedure.
App (extends built in Map<string,Layer>; implements Serializable)	
String name	The name of the application
constructor(Component vm, string name)	Stores passed data (uses vm as the application ViewModel), initializes fields.
Map findParent(string name)	Returns the parent for the requested by name children node using _findParent()
Map _findParent(string name, Map node)	A recursion step that controls if the node has a child with the specified name. If it has then returns this child, else continues recursion.
Object serialize()	Returns a serialized representation that can be used to restore this object.
void unserialize(Object image)	Uses the passed serialized representation to restore this object.
Layer (extends built in Map<string,Row>; implements Serializable)	
String name	The layer name.
Object desktop	Stores layout properties for desktop.
Object mobile	Stores layout properties for mobile.
Map<string,string> properties	Stores chosen reactive property values.
Map<string,string> propTypes	Stores reactive property types.
constructor(string name)	Stores the passed name, initializes fields.
Object serialize()	Returns a serialized representation that can be used to restore this object.
void unserialize(Object image)	Uses the passed serialized representation to restore this object.
RowCol	
String name	Stores the row or layer name.

Object desktop	Stores layout properties for desktop.
Object mobile	Stores layout properties for mobile.
Map<string,string> properties	Stores chosen reactive property values.
Map<string,string> propTypes	Stores reactive property types.
Row (extends built in Map<string,RowCol>, RowCol; implements Serializable)	
constructor(string name)	Stores the passed name, Initializes fields.
Object serialize()	Returns a serialized representation that can be used to restore this object.
void unserialize(Object image)	Uses the passed serialized representation to restore this object.
Col (extends RowCol, implements Serializable)	
Map<string,string> events	Stores changeable component event handlers.
string style	Stores changeable component style.
Map<string,string> slots	Stores changeable component slots.
constructor(Component vm, string name)	Stores passed name and component, initializes fields. Extracts properties and style from the component, generates empty event handlers, and calls extractSlots(), bind2way().
void extractSlots()	Extracts slot from component template.
bind2way(String properties[], Functions, functions)	Replaces property assignment with event emitting in functions.
Object serialize()	Returns serialized representation that can be used to restore this object.
void unserialize(Object image)	Uses passed serialized representation to restore this object.
AppExporter (implements TextExporter)	
vueURL	The URL to download Vue.js framework.
bsGridURL	The URL to download Bootstrap Grid.
string styles[]	Stores styles of used components

string code[]	Stores code of used components
string methods[]	Stores application methods and event handlers for used components
string heads[]	Stores generated include fields of the application and used components.
string names[]	Stores component names
constructor(App app, CSSExporter exporter)	Stores the app and the exporter, initializes fields
string export()	Returns generated application code using generate(),string templates, regular expressions, cssExporter and functionality provided by the Component class.
string generate()	Traverses application tree recursively using _generate() to process layers with their children.
string _generate(Node parent, boolean l1)	Recursion step that generates and returns html for the parent node. Populates heads, names arrays and uses genStyle(), genCode(), genHandlers(), genSlots() for additional processing ir parentNode is a Col instance. A root call has l1=true.
void genStyle(Col node)	Populates the styles array by generating style for the passed node.
void genCode(Col node)	Populates the code array by generating component code for the passed node..
void genHandlers(Col node)	Adds event handlers stored in the passed node to the methods array.
string genSlots(Col node)	Processes and returns slot code of the passed node.
CSSExporter (implements TextExporter)	
int desktopMinWidth	Value that is used in media query for generated desktop classes. Represents minimal width screen must have to apply these classes.
constructor()	Initializes fields.
string export()	Returns generated CSS.
LocalAppLibrary (extends LocalComponentLibrary)	

Object get(string name)	Unlike get in the LocalComponentLibrary does not deal with forking.
void set(Object serialized)	Unlike set in the LocalComponentLibrary does not deal with forking.
RemoteAppLibrary (extends RemoteAppLibrary)	
Object get(string name)	Unlike get in the RemoteComponentLibrary does not deal with forking.
void set(Object serialized)	Unlike set in the RemoteComponentLibrary does not deal with forking.
AppInstaller	
string apiURL	URL of the installation API.
string bsGridURL	URL to download (and cache by a service worker) Bootstrap Grid.
string vueURL	URL to download (and cache by service worker) Vue.js framework.
constructor(AppExporter exporter)	Saves passed exporter to use it for code generation, initializes fields.
Void install(App app)	Uses passed app to perform installation procedure. Uses findInclude() to find all static dependencies that must be cached.
Void findInclude(Node node, string include[])	Recursively finds include fields in the passed node and its children and populates the include array.

Table 16. Description of the application builder classes

8.3 Implementation

The following section contains remarks and nuances of the implementation process. The application builder uses exactly the same technologies as the component creator.

It also uses regular expressions for code processing. The table below contains a list of used regular expressions. It does not contain regular expressions that was described in the component creator and also are used in the application builder.

Regular expression	**Description**
/<slot.*name=["']\w+["'](.|\n)*?<\/slot>/ig	Extract slots from the template.
/name=["']\w+["']/i	Extract the name attribute of the slot.
/>(.|\n)*<\/slot>/ig	Extract the inner content of the slot.
/<slot.*>/ /:[a-z0-9_$]+/ig	Extract names of the slot’s bound attributes.
/this\.(property1|property2)\s*=\s*(.+?)(;|\n)/g	Find property assignment. And replace with 'this.$emit(\'update:$1\',$2);' (event emitting)
/(this\.)(component1|component2)(?=[\W]|$)/g	Finds component instance name identifiers and replaces with `this.$refs['$2']` (Vue.js special syntax)
/(\.[a-z0-9_$#\.\-]+)/ig	Finds CSS classes and replaces with '#'+node.name+' $1' (appends the component instance name as an id)
/(\W|^) (attribute1|attribute2) (\W|$)/ig	Finds slot attributes in the template and replace with '$1slotProps.$2$3' (referencing via specially created context)

Table 17. Used regular expressions (application builder)

By default, HTML elements handle mouse events and do not propagate them to the elements that locate below them. This behaviour is unwanted in case of layers, because top layers make impossible to handle mouse events by bottom layers. To overcome this, both layers and rows use pointer-events:none CSS property. This makes them transparent for mouse (pointer) events and components that locate below them can handle mouse events normally.

The workflow of the application code generation can be illustrated using these steps (this process takes place entirely inside the AppExporter’s export() method that uses the App instance as a data source. export() can be called by PreviewController or AppInstaller):

1. export() method creates empty arrays to store generated component data like styles, code, methods, include, names. Then it calls export() method to generate HTML and populate these arrays.

2. generate() is the entry point to traverse application tree recursively. It generates HTML code for layers and calls _generate(layer,true) to generate inner HTML.

3. _generate() method represents recursion step and if its second argument is true, then it generates additional enclosing row HTML (for the direct content of layers). If it is not, then it generates suitable HTML (uses layout parameters) for the node passed as the first argument that can be Row or Col instance. If the node is a Row instance then it calls _generate(node,false) to generate its inner HTML. If it is a Col instance, then it generates its HTML with slots using genSlots(col) and reactive component attributes. It also calls genCode(), genStyle(), genHandlers() to populate arrays, pushes value of the head property and the name property of the component into appropriate arrays.

4. genSlots() processes content of slots by putting it into a generated template tag with s specified slot-scope attribute. The content of slot is processed by prepending all ViewModel entries used by this slot (extracted during component loading in the Col class and putted into slotArgs map) with the value of the slot-scope attribute. This is required to transform the simplified syntax allowed by the application builder into the Vue.js scoped slots syntax.

5. genCode() generates component code using capabilities provided by Component with code that registers these components.

6. genStyle() prepends CSS classes with unique ids (component names are used) to differentiate between styles related to different instances of the some component.

7. genHandlers() populates array of functions using a event name prepended with a node name as the resulting handler name.

8. After generate() has finished its work, export() uses all data collected by the generate() to assemble the final HTML page using ES6 string template syntax. It generates application (root) ViewModel (Vue.js instance) using Component's generation capabilities. If generate() was called with the service worker argument (was called by AppInstaller), then it generates additional code required for PWA installation. It also searches for used component names in the code to replace them with the Vue.js $refs syntax.

9 Server-side implementation

The previous 2 chapters described implementation of the component creator and the application builder. But they were dedicated for the client side only. But the solution includes remote component library functionality and installation possibility. Both processes require a remote server. This chapter is dedicated to implementation of these processes.

9.1 Remote component library

The component library must allow CRUD operations with components. The RemoteComponentLibrary class has the following methods:

Object get(string name)	Returns a serialized representation of the component.
void set(Object serialized)	Inserts a serialized component.
void delete(string name)	Deletes a component by his name.
string[] all()	Returns all component names.

Table 18. RemoteComponentLibrary methods

Like the local component library, the remote one stores serialized component representations without any knowledge of their inner structure. The minimal implementation must allow to work with key-value pairs. This can be done using a file system, a document-oriented database like MongoDB, or a SQL database like MySQL This requires quite a little server side logic and can be implemented using different technologies. For example, implementation from the prototype developed in this thesis uses PHP with MySQL.

More important is to define a web API of the remote component library. This will allow to connect to different component library implementations that implement this API. The simplest way is to use the JSON format to transfer a serialized component. For other data and arguments, the similar JSON format can be used. Transferring JSON data requires the POST HTTP method and allowing requests from other servers (origins) requires the

Access-Control-Allow-Origin CORS (Cross-origin Resource Sharing) HTTP header to be set.

The API request is always POST HTTP method. Its body contains a JSON object with the method field that defines a concrete action. Other fields contain method arguments. The response is also in the JSON format. The following table contains the remote component library API description.

Request POST body	Response body	Actions
{“method”:”set”, ”name”:”<component name>”, “component”:<serialized representation>}	None.	Save the passed component using its name. If the component with this name already exists, then replace it.
{“method”:”get”, ”name”:”<component name>”}	Serialized representation	Find a component with the passed name and return it or an empty object if there is no component with this name.
{“method”:”delete”, ”name”:”<component name>”}	None	Delete a component with the passed name. If there is no component with this name then do nothing.
{“method”:”all”}	An array of component names..	Find all component names currently stored in the component library and return an array of them.

Table 19. Remote component library API

The implementation that was discussed in the previous chapter also has a RemoteAppLibrary class that allows CRUD operations with apps. This class uses the API exactly similar with the API previously described. The only one difference is that instead of serialized component representations this API uses serialized application representations.

9.2 Installation

To be installed a PWA require service workers to be transferred via HTTPS. The certificates that are used during HTTPS session cannot be self-signed, they must be signed by the 3-rd party (certificate authority). This is another reason to use a remote server. The install() method of the AppInstaller class that was created for installation purposes uses web API for PWA installation. Like the remote component library API, the installation API uses HTTPS POST method and stores all data and arguments inside a request body in the JSON format. There is no a response body. The table below describes fields of the request POST body.

Field	**Type**	**Description**
name	string	The name of the application
app	string	Generated application code
cache	array of strings	URLs of static assets used by the application and its components
manifest	object like {“name”:”app”,…}	Contains a configurable manifest part.

Table 20. Installation API request fields description

The installation process consists of the following actions: The application builder makes a request to the server using the previously described APIs. It uses the AppExporter class to generate code with additional PWA specific headers and service worker registration code. The server takes data from the request and creates a folder with the name as was specified in the name field. Then it writes the application code into the file named “index.html”. It uses the cache field to generate a service worker with the “cache falling back to the network” policy and the manifest field to generate a manifest. It uses a random version to force application cache to be updated when the new service worker is uploaded. Then it saves both files into the previously created folder with the name “sw.js” and “manifest.json” respectively. It also copies an icon to this folder with the name “icon192.png”. Of course another names can be used, but the application builder must be aware about them. Then the application builder opens the created “index.html” in the new window. The application builder uses a random URL argument to overcome browser cache when it requests the page, the manifest and the service worker. This allows to get the actual version of the application. The next actions are controlled by the browser. It controls PWA requirements and shows an installation banner to the user. The following sequence diagram shows the previously described installation process.

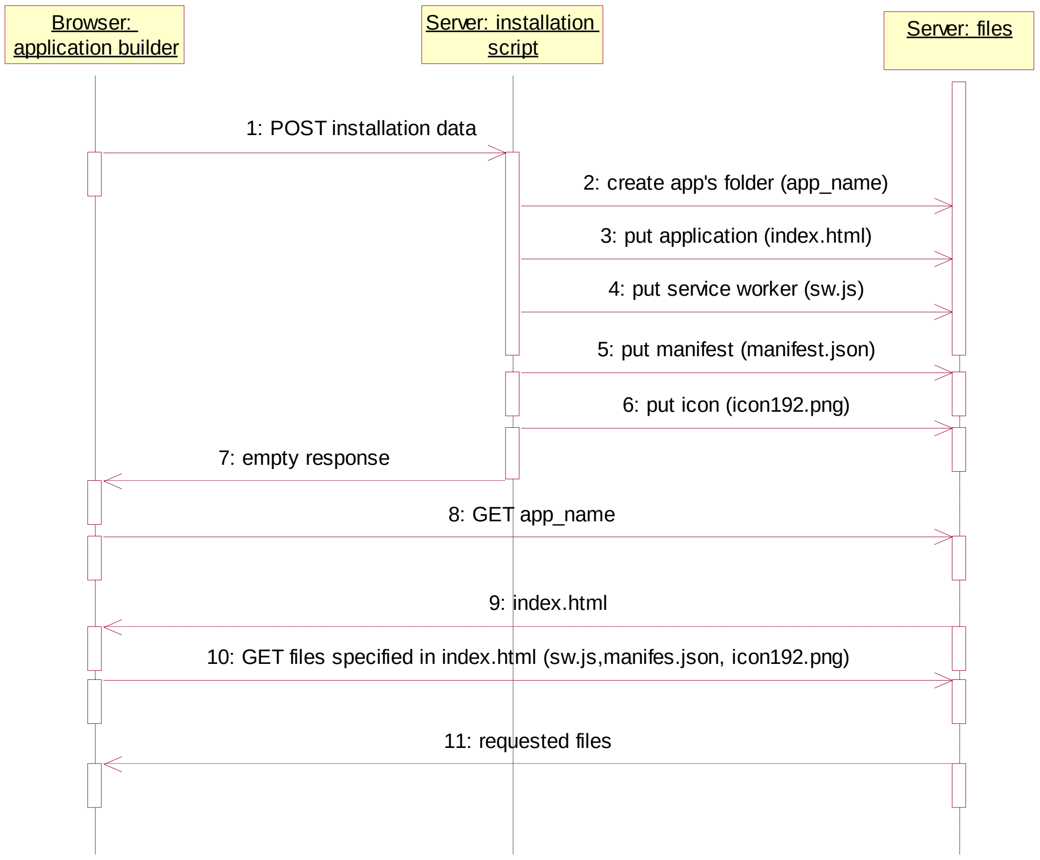

Browser: application builder
Server: installation script
Server: files
1: POST installation data
2: create app's folder (app_name)
3: put application (index.html)
4: put service worker (sw.js)
5: put manifest (manifest.json)
6: put icon (icon192.png)
7: empty response
8: GET app_name
9: index.html
10: GET files specified in index.html (sw.js,manifes.json, icon192.png)
11: requested files

Figure 16. The PWA installation sequence diagram

10 Evaluation and results

The aim of this chapter is to evaluate the solution that was implemented during this thesis. Firstly, it defines criterions that can be used to evaluate the solution. They are made based on the problem statement: Simplicity in the component-based PWA creation using Vue.js MVVM framework. The conceptual solution chapters already have analytical evaluation and expected results, but existence of the real prototype allows to conduct more deep and “close to the real life” evaluation in some aspects. In addition to this, the implemented prototype allows to use test methods to receive concrete quantitative results in some aspect like size of hand written code or number of actions.

10.1 Analytical evaluation

As the title says, the aim of this master thesis is to simplify process of the component-based PWA development using Vue.js MVVM framework. This title can be disassembled into keywords: simplification, Vue.js, component-based, PWA development. Firstly, it is important to control that the developed solution relates to these keywords and as a result relates to the problem statement. The table below shows this.

Keyword	**Solution**
Component based	Vue.js is designed to develop reusable GUI components. The component creator was specially developed to assist with Vue.js components creation.
Vue.js	The component creator produces Vue.js components and allows to use its concepts during development. Application builder uses Vue.js as the application’s view model provider. It also uses Vue.js components and allows their customization using Vue.js concepts.
PWA development	Additional functionality was specially added to transform developed application into PWA. Application install banner appearance shows that the browser recognizes the web application as a PWA.
Simplification	When the component creator and the application builder were developed, the rules of simplicity from the Table 3 were used.

Table 21. Basic evaluation

The table above shows that the solution relates to the gap defined in the problem statement. The solution was specially targeted to the Vue.js component based framework and additional functionality was added to fulfil PWA requirements. These facts do not need future discussion.

But the most important part of the solution is the simplicity requirement. This requirement affects the whole process: component creation, component management, application building and PWA producing. This is how the developed solution differs from other solutions. Chapters 3-6 end with evaluation of the conceptual solution that was developed during these chapters. This evaluation was based only on theory and another words can be treated as an expected result. Existence of the real prototype allows to control how it works in the real world. Some parts of the conceptual solution work as expected, but some other parts are not as good in the real life as in the theory. The following table shows parts of the solution that met some problems or obstacles to achieve the expected result. Cons that are shown there was discovered only after prototype implementation and was not obvious during conceptual solution development. Parts of the conceptual solution that are not shown in this table work as expected and do not require additional remarks.

Solution	**Result (conceptual)**	**Result (real)**
Generate a component skeleton code providing editing fields only for the required parts that define a View and a ViewModel of the component.	Amount of handwritten code decreases. A developer focuses only on the required parts. Frees the developer from knowing the automatically generated parts	Developers are used to use text editors for their programming purposes. Some of them prefer to see and edit source code entirely, not only the gaps that are really matter. And this approach is unfamiliar for them.
Group methods and computed properties into lists with add/remove/edit possibilities.	Reduces amount of code editing actions that must be taken to create or delete a function.	As was said before, developers are used to use text editors for their programming purposes. This approach is unfamiliar for programmers.
Component creator: Provide an automatically updating preview with possibilities to show errors and exceptions.	No need to write a html page and refresh the browser after editing manually. Shows errors and exceptions inside the preview without need to open a console window.	For mobile browsers that do not have console, this is the only one way to show errors. But desktop browsers that have dev tools have better possibilities for debugging.

Provide an interactive layout view consisting of tabs (layers) , rows and columns (other rows or components) with possibilities to add/delete rows and add/delete components.	Easy and intuitive Bootstrap Grid layout composition without code writing	Some users want to have more advanced layout builder. Like interacting with layout directly (like drag resizing and moving), drag and drop functionality. Instead of selecting and editing properties.
Use 2 sets of layout properties: for the desktop and for the mobile.	Possibility to use one layout for the desktop and another one for the mobile using only grid layout parameters without creating another view hierarchy.	Only 2 breakpoints do not allow to use all screen space effectively on wide screens.
Automatically generate empty event handlers and show a list of them.	The developer has a clear picture about component events and focuses only on the event handler body.	There is no way to know exactly arguments of the event handler (JavaScript can call any function with any arguments). This can be frustrating when generated handler has a wrong signature.
Inject children components directly into the parent's ViewModel using the name of the component instance as a field name.	More convenient and intuitive way to access components without dealing with the specific Vue.js syntax.	The name of the component instance must be valid JavaScript identifier that is additional requirement.
Provide a separate style editing field.	Possibility to use pseudo-elements. Better separation of concerns. Visually more readable and easier to follow. Can style every component instance without explicit id.	To differentiate between component instances, the name of the component instance must be a valid CSS id that is additional requirement.
Application builder: Use double preview mode: The automatically updating interactive layout preview and the manually updating final preview.	The developer has a clear idea about his application layout without additional actions. If he wants to see the final result the only one action he needs is to press the “update” button.	Component height is not known during layout preview. This makes it not so clear as expected. No way to see desktop preview on mobile screen without rotation. Desktop browsers have device screen emulators that have more possibilities to test

		application in the mobile viewport.
Provide the developer with the component library where he can save his components and open them for editing or using in future.	The developer always has easy-accessible list of components and can use/open/save them in one click.	Requires additional functionality like grouping, filtering, sorting as number of components grows.
Possibility to install the application.	Developer gets his fully working application in one click. The application exists entirely separately from the ecosystem developed in this thesis.	The browsers show the installation banner only for the first time, then disables it for the period of time. To install the application the user must select an appropriate option from the browser menu.
Inject children components directly into the parent's ViewModel using the name of the component instance as a field name.	More convenient and intuitive way to access components without dealing with the specific Vue.js syntax. As a result, error probability also decreases.	When the component is hidden using conditional rendering, it also is missing from the Vue.js $refs property and cannot be accessed at all. Conditional rendering results in the component destruction.

Table 22. Conceptual vs real solution evaluation

Also, when the component builder was made in the form of the PWA, it was discovered that PWAs cannot trigger the installation process for other PWAs and the prototype was made in the form of a web application but not a PWA.

The chapters related to the conceptual solution have textual description of the syntax that can be used to make development more simple. The table below contains concrete examples of its application. It has a task example in the first column, then this task is expressed using simplified syntax in the second column and generated Vue.js syntax (that is used in the traditional approach) in the third column.

Task	Simplified syntax	Vue.js syntax
Property definition	String name ‘something’	name:{ type:String, default:’something’ }
ViewModel variables definition	this.variableName	function(){ return {variableName:undefined}; }
two-way data binding	this.propName=’value’	this.$emit(‘update:propName’,’value’);
Using ViewModel variables inside a slot	{{variableName}}	<template slot-scope=”slotProps”> {{slotProps.variableName}} </template>
Use children components	this.child	this.$refs(‘child’);

Table 23. Simplified syntax examples

10.2 Experimental testing

This is very hard to measure simplicity. The majority of the parts that form simplicity cannot be measured rigorously using concrete numbers. One cause of this is that simplicity perception is quite subjective. It can be measured roughly using qualitative methods. This was made in the previous part of this chapter: expected outcomes of the conceptual solution was controlled using implemented prototype.

But there still exists very important criterion that can be measured using quantitative methods. This is the number of actions that the developer must perform to get his final application. This affects the following parts of the simplicity definition: time saving and a learning curve. The learning curve is affected because reducing the number of actions also reduces amount of information that must be learned to effectively use the tool.

The number of action will be evaluated using the created prototype in action. As the number of actions the amount of hand written code is implied. As a testing task will be created a small image gallery that contains NASA missions with their description. When the user selects a mission, content of the gallery is updated (contains thumbnails). When he clicks the thumbnail, application shows a big version of the picture with description that can be closed by clicking it. The developing application also requires creation of 2 components: a simple navigation bar that will display mission names and a gallery that will display pictures with additional information.

The appendix 1 shows code examples of the generated navigation bar and the gallery components. These code examples show that the developer really focuses only on the code that really matter (his own code) and amount of side code is minimized. When components are ready, the application can be built from them in graphical mode and amount of the additional hand written code is very small. The appendix 2 shows code examples of generated assets that are required to form a PWA: index.html, sw.js and manifest.json.

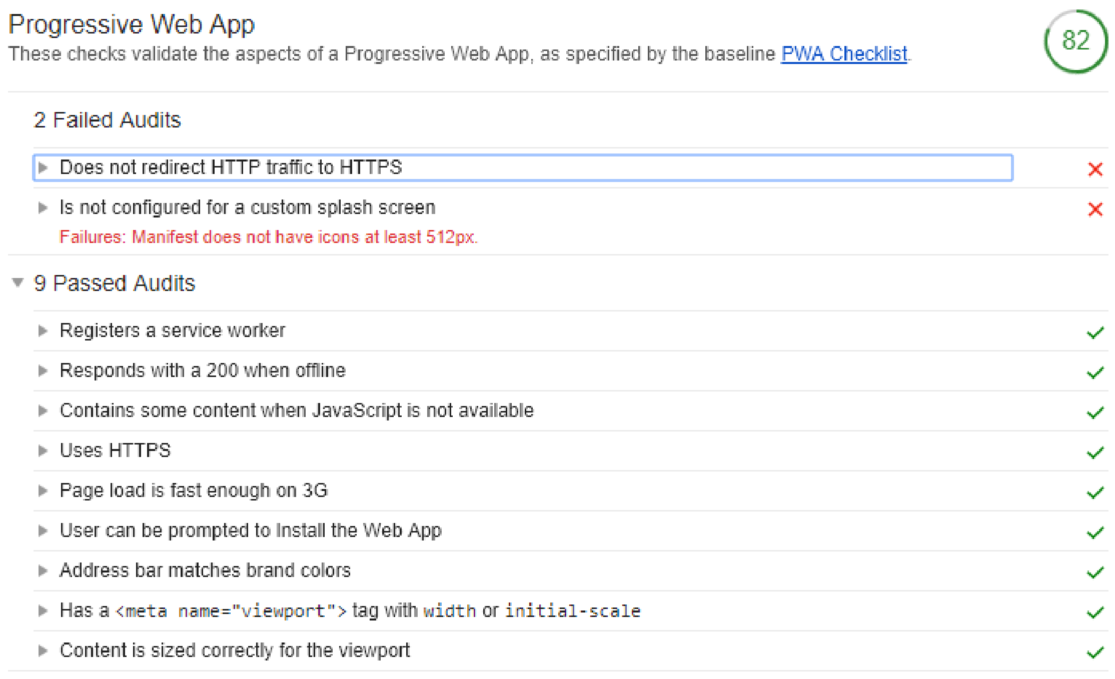
The NASA gallery test application does not use all possibilities of the developed prototype, but gives a clear picture about hand written code reduction. It shows that the biggest amount of hand-written code locates inside components. When components are created, their using does not require much additional code (moreover, the code fraction is very small). But on the other hand, the application developer can customize components using styles and slots that supports the last rule of simplicity (simplicity and complexity need each other). The service worker and the application manifest also consist of the generated code almost entirely.

Different applications require different amount of code. Complex application has greater fraction of the hand written code, but the small applications benefit more. The table below contains an approximate hand written code fraction based on the testing application.

Part	**Fraction**
Component	~70%
Application	<10%
Service worker	0% (requires to specify values only)
Manifest	0% (requires to specify values only)

Table 24. An approximate hand written code fraction

In additional to this, generated application was tested using Lighthouse audit. It is designed to test PWAs and can be used from Chrome DevTools. The figure below shows its results for the test application described above. HTTP to HTTPS redirection must be done on the server side. The solution developed in this thesis deals with the client side only. Splash screen issue can be fixed using large (512+px) icons. As developed solution is only the prototype this is not very important. Other criterions were fulfilled, and the test application can be treated as a PWA.


Progressive Web App

These checks validate the aspects of a Progressive Web App, as specified by the baseline PWA Checklist.

82

2 Failed Audits

- Does not redirect HTTP traffic to HTTPS ✗
- Is not configured for a custom splash screen ✗
  Failures: Manifest does not have icons at least 512px.

9 Passed Audits

- Registers a service worker ✓
- Responds with a 200 when offline ✓
- Contains some content when JavaScript is not available ✓
- Uses HTTPS ✓
- Page load is fast enough on 3G ✓
- User can be prompted to Install the Web App ✓
- Address bar matches brand colors ✓
- Has a `<meta name="viewport">` tag with `width` or `initial-scale` ✓
- Content is sized correctly for the viewport ✓

Figure 17. Lighthouse audit results

10.3 Future improvements

The implemented solution is only a prototype. But it shows that the conceptual solution can be released and it positively affects simplicity in general. But the Table 22 shows that this solution has some drawbacks. To effectively apply the developed solution in practice, majority of these drawbacks must be fixed. This can be done in the future iterations when the conceptual solution can be updated using experience collected from the prototype implemented during this thesis. Then the another prototype can be built based on the conceptual solution and evaluation can be repeated. After two or more these iterations a finished result can be achieved. The future iterations can also improve the existing parts like using more flexible regular expressions. And after adding additional non-functional requirements a ready to wide use final product can be released.

11 Summary

The aim of this thesis was to simplify a process of building PWAs using Vue.js MVVM framework. The thesis shows that an effective problem solution requires development of an integrated solution in the form of a toolchain or an ecosystem. This solution covers Vue.js component creation, application building using these components, components management and transformation the resulting application into a PWA.

The solution consists of 2 parts: a theoretical conceptual solution and its practical implementation. During the theoretical part, the standard way was analysed to find parts that can be improved orienting to the stated definition of simplicity. This part allows to conduct theoretical evaluation only. To achieve real results, the conceptual solution was implemented in the form of a prototype. The prototype was made in the form of a web application to share the execution environment with applications it helps to create. The prototype existence allows to conduct evaluation more precisely. It helped to find drawbacks of the conceptual solution that was not obvious during its evaluation phase. Additionally, it allows to apply experimental testing methods using concrete test examples to measure quantitative properties of simplicity.

The evaluation shows that the developed solution helps to achieve stated results in general. The solution helps with simplification of the PWA development process. But the evaluation also shows that the implemented prototype has some drawbacks that negatively affect simplicity. These drawbacks can be fixed during the future development.

References

[1] "Chrome OS - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Chrome_OS. [Accessed 15 March 2018].

[2] "5 Best JavaScript Frameworks in 2017," [Online]. Available: https://hackernoon.com/5-best-javascript-frameworks-in-2017-7a63b3870282. [Accessed 7 March 2018].

[3] "The Good and the Bad of Angular Development," [Online]. Available: https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/. [Accessed 15 March 2018].

[4] "node.js - Google Trends," [Online]. Available: https://trends.google.com/trends/explore?date=2010-09-17%202017-09-17&q=node%20js. [Accessed 15 March 2018].

[5] A. Hevner, S. March, J. Park and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly,* vol. 28, pp. 75-105, 2004.

[6] "Progresseve Web Apps," [Online]. Available: https://developers.google.com/web/progressive-web-apps. [Accessed 17 February 2018].

[7] "Service Workers: an Introduction," [Online]. Available: https://developers.google.com/web/fundamentals/primers/service-workers/. [Accessed 17 February 2018].

[8] "Measure Performance with the RAIL Model," [Online]. Available: https://developers.google.com/web/fundamentals/performance/rail#goals-and-guidelines. [Accessed 17 February 2018].

[9] "Can I use?," [Online]. Available: https://caniuse.com/. [Accessed 17 February 2018].

[10] D. Kardys, "Modular Web Design: Designing With Components," [Online]. Available: https://blog.wsol.com/modular-web-design-designing-with-components. [Accessed 25 February 2018].

[11] K. J., Introducing Bootstrap 4, Apress, 2016.

[12] M. Moskala, "MVC vs MVP vs MVVM vs MVI," [Online]. Available: https://academy.realm.io/posts/mvc-vs-mvp-vs-mvvm-vs-mvi-mobilization-moskala/. [Accessed 3 March 2018].

[13] "JavaScript + jQuery Design Pattern Framework," [Online]. Available: http://www.dofactory.com/products/javascript-jquery-design-pattern-framework. [Accessed 3 March 2018].

[14] "Vue.js Guide," [Online]. Available: https://vuejs.org/v2/guide/index.html. [Accessed 3 March 2018].

[15] J. Maeda, The Laws of Simplicity, The MIT Press, 2006.

[16] V. Gupta, "INLINE VS INTERNAL VS EXTERNAL CSS," [Online]. Available: https://vineetgupta22.wordpress.com/2011/07/09/inline-vs-internal-vs-external-css/. [Accessed 4 March 2018].

[17] "Single File Components," [Online]. Available: https://vuejs.org/v2/guide/single-file-components.html. [Accessed 4 March 2018].

[18] "TIOBE Index for February 2018," [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed 4 March 2018].

[19] "Top 5 Most Popular CSS Frameworks that You Should Pay Attention to in 2017," [Online]. Available: https://hackernoon.com/top-5-most-popular-css-frameworks-that-you-should-pay-attention-to-in-2017-344a8b67fba1. [Accessed 11 March 2018].

[20] "Viewport Sizes," [Online]. Available: http://viewportsizes.com/. [Accessed 15 March 2018].

[21] "Caching Files with Service Worker," [Online]. Available: https://developers.google.com/web/ilt/pwa/caching-files-with-service-worker. [Accessed 20 March 2018].

[22] "Web App Install Banners," [Online]. Available: https://developers.google.com/web/fundamentals/app-install-banners/. [Accessed 24 March 2018].

[23] "Code Mirror: Simple Mode Demo," [Online]. Available: https://codemirror.net/demo/simplemode.html. [Accessed 5 April 2018].

[24] E. Kitamura, "Working with quota on mobile browsers," [Online]. Available: https://www.html5rocks.com/en/tutorials/offline/quota-research/. [Accessed 19 March 2018].

[25] "IndexedDB API - Web APIs | MDN," [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API. [Accessed 5 April 2018].

[26] "Composite pattern - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Composite_pattern. [Accessed 16 April 2018].

Appendix 1 – An code example generated by the component creator

The blue colour indicates generated code and the red one indicates hand-written code.

The code of the navigation bar component:

```
<template><div>
        <div class="bar" style="display:flex; flex-wrap:wrap">
        <div v-for="item in items" class="item" @click="onclick(item)">
        <slot name="item" :item="item">{{item}}</slot>
</div></div>
</div></template>
<style>
        .bar{background:#222222;padding:12px;}
        .item{color:lightgray;cursor:default;
        font-size:24px;margin:0 10px 0 10px;
        text-align:center;}
        .item:hover{color:white;}
</style>
<script>
export default{
        props:{
                items:{type:Array,default:[]}
        },
        methods:{
                onclick(entry){this.$emit('click_item',entry)}
        },
}
</script>
```

The code of the gallery component:

```
<template><div>
<div style="display:flex; flex-wrap:wrap">
<div class="image-container" v-for="item in items" @click="onclick(item)">
<slot name="image" :item="item">
```

```
<img :src="item.src" class="image-img">
<div class="image-title">{{item.title}}</div>
<div class="image-description"></div>
</slot></div></div>
</div></template>
<style>
       .image-img{width:180px;height:150px;object-fit: cover;}
       .image-title{color: gray;font-size: 16px;font-weight:500;}
       .image-description{color: gray;font-size: 14px;}
       .image-container{background: white;
       margin:2px;border:1px solid gray;
       border-radius: 5px;padding: 5px;}
</style>
<script>
export default{
       props:{
              items:{type:Array,default:[]}
       },
       methods:{
              onclick(item){this.$emit('click_item',item);}
    },
}
</script>
```

Appendix 2 – An code example generated by the application builder

The blue colour indicates generated code and the red one indicates hand-written code. Yellow indicates the Model part of the MVVM pattern that is not a responsibility of this thesis.

The code of the generated index.html:

```
<html><head>
<link rel="manifest" href="manifest.json?0.9745235701368631">
    <meta name="mobile-web-app-capable" content="yes">
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta name="application-name" content="NasaGallery">
    <meta name="apple-mobile-web-app-title" content="NasaGallery">
    <meta name="msapplication-starturl" content="index.html">
    <link rel="icon" href="icon192.png">
    <link rel="apple-touch-icon" href="icon192.png">
    <title>NasaGallery</title>
<script>
if ('serviceWorker' in navigator)
    navigator.serviceWorker.register('sw.js');
function showError(at, err, line) {
  parent.postMessage({version:Vue.version,at,err,line},"*");
}</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.5.13/vue.min.js"></script>
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.0.0/css/bootstrap-grid.min.css">
<meta name="viewport" content="width=device-width, initial-scale=1">
<style>
.position-absolute {position: absolute !important;}
.position-fixed {position: fixed !important;}
.m0 {margin-left:0;margin-right:0;}
.p0 {padding-left:0;padding-right:0;}
.h100 {height:100%;}
.e0 {pointer-events:none;}
.e1 {pointer-events:auto;}
body {margin:0px;}.pl0{padding-left:0rem}
#thumbnails .image-img{width:180px;height:150px;object-fit: cover;}
#thumbnails .image-title{color: gray;font-size: 16px;font-weight:500;}
#thumbnails .image-description{color: gray;font-size: 14px;}
#thumbnails .image-container{background: white;margin:2px;border:1px solid gray;
border-radius: 5px;padding: 5px}
#navigation .bar{background:#222222;padding:12px;}
#navigation .item{color:lightgray;cursor:default;font-size:24px;margin:0 10px 0
10px;text-align:center;}
#navigation .item:hover{color:white;}
#mission .image-img{width:100%;object-fit: cover;}
#mission .image-title{color: gray;font-size: 16px;font-weight:500;}
#mission .image-description{color: gray;font-size: 14px;}
#mission .image-container{background: white;margin:2px;border:1px solid gray;
border-radius: 5px;padding: 5px;max-height:100vh;overflow:auto;}
</style></head>
<body style="background:#ffffff">
<div id="app" class="e0"><div class="container-fluid p0 position-absolute" ><div
class="row m0"><div class="col-sm-12 col-12 offset-sm-0 offset-0 align-self-sm-start
align-self-start pl0 pt20 pr0 pb0 plsm0 ptsm20 prsm0 pbsm0" ><div class="row
m0"><gallery class="col-sm-12 col-12 offset-sm-0 offset-0 align-self-sm-start align-
self-start pl0 pt0 pr0 pb0 plsm0 ptsm0 prsm0 pbsm0 e1" id="thumbnails"
```

```
@click_item="thumbnailsclick_item" :items.sync="images" ref="thumbnails"><template
slot="image" slot-scope="slotProps"><img :src="slotProps.item.src" class="image-img">
<div class="image-title"> {{slotProps.item.data.date_created}} </div>
<div class="image-description"></div></template></gallery>
</div></div></div></div><div class="container-fluid p0 h100 position-fixed" ><div
class="row h100 m0"><div class="col-12 p0 align-self-start"><div class="row m0"><div
class="col-sm-12 col-12 offset-sm-0 offset-0 align-self-sm-start align-self-start pl0
pt0 pr0 pb0 plsm0 ptsm0 prsm0 pbsm0" ><div class="row m0"><navbar class="col-sm-12
col-12 offset-sm-0 offset-0 align-self-sm-start align-self-start pl0 pt0 pr0 pb0
plsm0 ptsm0 prsm0 pbsm0 e1" id="navigation" @click_item="navigationclick_item"
:items.sync="missions" ref="navigation"><template slot="item" slot-scope="slotProps">
{{slotProps.item}}  </template></navbar></div></div></div></div></div></div><div
class="container-fluid p0 h100 position-fixed" ><div class="row h100 m0"><div
class="col-12 p0 align-self-center"><div class="row m0"><div class="col-sm-12 col-12
offset-sm-0 offset-0 align-self-sm-start align-self-start pl0 pt0 pr0 pb0 plsm0 ptsm0
prsm0 pbsm0" ><div class="row m0"><gallery class="col-sm-8 col-12 offset-sm-2 offset-
0 align-self-sm-start align-self-start pl0 pt0 pr0 pb0 plsm0 ptsm0 prsm0 pbsm0 e1"
id="mission" @click_item="missionclick_item" :items.sync="currentMission" v-
if="missionShown" ref="mission"> <template slot="image" slot-scope="slotProps"> <img
:src="slotProps.item.src" class="image-img">
<div class="image-title"> {{slotProps.item.data.title}} </div>
<div class="imagedescription"> {{slotProps.item.data.description}} </div> </template>
</gallery></div></div></div></div></div></div>
<script>
app=new Vue({el: '#app',
data: function(){return {
missions:undefined,
missionShown:undefined,
currentMission:undefined,
images:undefined
}},
created(){
        this.loadData('apollo');
        this.missions=['apollo','aqua','barrel','dawn'];
        this.missionShown=false;
        this.currentMission=[];
},
methods:{
loadData(query,n=20){
        fetch(`https://images-
api.nasa.gov/search?q=${query}&media_type=image`).then((response)=>{
        response.json().then((data)=>{
        this.images=data.collection.items.map(d=>{return
        {src:d.links[0].href,data:d.data[0]}}).slice(0,n);
})});},
thumbnailsclick_item:function(item){this.currentMission=[item];
       this.missionShown=true;},
navigationclick_item:function(item){this.loadData(item);},
missionclick_item:function(){this.missionShown=false;}
}});</script></body></html>
```

The code of the generated sw.js:

```
VERSION='v1591801207';
this.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open(VERSION).then(function(cache) {
      return cache.addAll(['manifest.json','icon192.png',
'index.html',
'https://cdnjs.cloudflare.com/ajax/libs/vue/2.5.13/vue.min.js',
'https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.0.0/css/bootstrap-
grid.min.css',
https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js,
]);}));});
this.addEventListener('activate', function(event) {
  event.waitUntil(caches.keys().then(function(keyList) {
    return Promise.all(keyList.map(function(key) {
      if (key!=VERSION) {
        return caches.delete(key);
      }}));}));});
this.addEventListener('fetch', function(event) {
  event.respondWith(caches.match(event.request).then(
    function(response) {
      return response || fetch(event.request);
    }));});
```

The code of the generated manifest.json:

```
{"name": "NasaGallery",
"short_name": "NasaGallery",
"description": "This is a test app!",
"background_color": "#61e9db",
"theme_color": "#ffffff",
"start_url": "index.html",
"display": "standalone",
"orientation": "any",
"icons": [{
      "src": "icon192.png",
      "sizes": "192x192",
      "type": "image/png"
  }]}
```

Appendix 3 – Source code of the prototype

Git repository:

https://gitlab.cs.ttu.ee/Sergei.Malosev1/master-thesis

SSH clone URL:

git@gitlab.cs.ttu.ee:Sergei.Malosev1/master-thesis.git