Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Website builder - technical design and prototype

Peter Tóth

Supervisor: Bc. Petr Hurťák
May 2019

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Tóth** Jméno: **Peter** Osobní číslo: **453223**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Website builder – technický návrh a prototyp

Název bakalářské práce anglicky:

Website builder – technical design and prototype

Pokyny pro vypracování:

1. Describe a website builder and compare to different solutions. Study the use cases of browser-based website builders. Define the scope of the project.
2. Provide a technical design consisting of:
1. Description of basic system components: client and server application
2. Domain object model
3. Specification of used frameworks and critical dependencies
4. Analysis of the persistence layer
5. Analysis of the proposed user interface for website edition such as adding, modifying and removing page content blocks like text, images, buttons etc
3. Build a prototype of the system using modern Javascript technologies showcasing features specified in the project scope
4. Discuss suitable scaling approaches and performance bottlenecks such as website caching, load balancing, database replication and static file management & hosting

Seznam doporučené literatury:

[1] Learning Node - Shelley Powers - ISBN 1449323073
[2] Vue.js: Up and Running: Building Accessible and Performant Web Apps -
Callum Macrae - ISBN 1491997249
[3] Practical Object-Oriented Design in Ruby: An Agile Primer - Sandi Metz - ISBN
0321721330

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Bc. Petr Hurťák, katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **29.01.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Bc. Petr Hurťák podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgements

I would like to thank my supervisor Mr. Petr Hurťák for providing guidance and sharing his valuable expertise throughout writing the thesis.

I would also like to thank my friend Štefan Štefančík for sharing his opinions on the user experience and aesthetic aspects of the prototype.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 24, 2019

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 24. května 2019

Abstract

Website builders are tools that enable people create websites without the need to code them manually. The goal of this thesis is to provide an introduction to the problem of website builders and analyse, design & implement a prototype of a website builder using modern web technologies. Finally, provide reasoning for technology considerations and an overview of the best practices.

Keywords: website builder, javascript, vuejs, nuxt, server side rendering, component, tree, user interface

Supervisor: Bc. Petr Hurťák

Abstrakt

Tvůrce webů jsou nástroje které umožňují lidem vytvořit webové stránky bez potřeby psát kód manuálně. Cílem této práce je uvést do problematiky tvůrců webů a analyzovat, navrhnout & implementovat protoyp tvůrce webů použitím moderních webových technologií. Nakonec, poskytnout odůvodnění vybraných technologií a přehled osvědčených postupů.

Klíčová slova: tvůrce webů, javascript, vuejs, nuxt, server side rendering, komponenty, stromová struktura, uživatelské rozhraní

Překlad názvu: Website builder - technický návrh a prototyp

Contents

Figures

Tables

Chapter 1
Introduction

1.1 Introduction

Ever since first websites appeared in early 1990s, people were coming up with tools to build them without needing to write HTML(Hypertext Markup Language) code manually. Starting with Microsoft FrontPage in 1995, being essentially a desktop HTML WYSIWYG(What you see is what you get) editor, rivalled by Adobe Dreamweaver which later established itself as industry leader in web development tools. The popularity of website builders peaked in late 1990s when GeoCities was launched and enabled its users to create fully hosted static websites from within their web browser.

Website builders typically fall into two categories:

- **Offline**

  Offline website builders are desktop applications that run on user computer, allowing users to edit the websites on their computer and then export the websites as a piece of HTML, CSS(Cascading Style Sheet) and Javascript code.

- **Online**

  Online website builders are proprietary tools provided by web hosting companies.[4] They run as web applications accessible through the web browsers and allow users to create and edit websites through their web user interface.

  Many online website builders also let their users to export the code of the built websites but key differentiation is that online website builders can also host the websites created by them, removing the complexity of hosting a website from the user.

In this thesis, we will explore the latter category by designing and implementing a prototype of an online website builder.

1.2 Motivation

Currently, there are many solutions that enable users to create websites without the need to code them or have any prior web development skills. They vary in implementation, features, platform, language and ultimately the end user needs.

This thesis serves an experiment to develop a website builder which is easily extensible by new features and uses modern web technologies. On of the goals is to be able to host the websites built by the website builder, meaning we'll be able to render them on the server.

The key motivation factors for this thesis are following:

- Develop a highly versatile website builder with full WYSISYG interface
- Explore new modern web technologies
- Utilize the prototype as a software product in the future

1.3 Goals

This thesis is about implementing a prototype of a website builder based on an analysis and a technical design. Its aim is also to provide a theoretical background on the topic and eventually discuss some technology considerations.

The chosen workflow has been divided into three steps:

1. **Introduce to the problem of website builders**

   First, we will define some technical terms and explain what the website builders are.

2. **Design and build a proof of concept of a website builder**

   In this step we'll analyse the problem to be able to come up with application specification followed by the analysis of the prototype implementation

3. **Discuss the design decisions and technology considerations**

   Lastly, we'll go through the rest of important topics such as best practices, scaling or security.

Chapter 2
Analysis

This chapter describes the technical design behind the prototype of our website builder. It provides an introduction to the domain object model and reveals how the application is going to work under the hood.

2.1 Technical design

2.1.1 Domain object model

When we're speaking about web pages, we typically speak of HTML documents that make up the markup of a web page. These documents consist of HTML DOM(Document object model) elements which are organised in a graph data structure, more specifically the tree data structure. A tree has a single root node that outer spans its children nodes. Every node that has at least one child is a root of its sub-tree. Nodes with no children are called leaves and they are found at the furthest depths of the tree.

Let's imagine a simple HTML markup like this:

```
<header>
  <div class="container">
    <h1>Title</h1>
    <img src="logo.png"/>
  </div>
</header>
```

If we want to build a website builder, we ultimately need to store some state of a markup in a persistent storage so it can be retrieved for later use. The simplest approach would be to turn the HTML document into a string value and store it as text field in the data stores.

This is good enough if we only want to read the state but to enable for addition, deletion and modification of the markup, we need to eventually change the state too.

Possibly, we can assume it's feasible to retrieve the state as a HTML markup in a form of a string, parse it to HTML DOM, perform modifications and repeatedly store it as a string value.

For our use case, the mentioned approach is not good for us due to these main reasons:

- **Markup is tied to implementation, not semantics**

  Let's look at an example markup of a simple row element:

  ```
  <div class="row">
    <div class="column">
      one
    </div>
    <div class="column">
      two
    </div>
    <div class="clearfix"></div>
  </div>
  ```

  We can see the markup contains the `<div class="clearfix"></div>` element. This is a common technique for fixing space around the elements that float. However, when the implementation changes with an advent of new technology or requirements, we would have to take special measures to replace the old markup stored as a text with the new one.

- **Validation and security issues**

  In case raw HTML is persisted, it needs to be validated against script injection and malicious content which can be utilised in `<iframe>` or `<script>` HTML elements.

- **Coupling with HTML DOM API(Application programming interface)**

  When our domain logic is tightly coupled with another domain logic we might lose a portion of flexibility. Additionally, changes on the foreign domain logic can result in breaking our application's domain logic.

Instead storing the markup state as a text of HTML document, we could store the markup in a structure like this:

```
element: Row
settings:
  clear: true
children:
  element: Column,
  element: Column
```

and try to think how we can map it to HTML like in the example before 2.1.1.

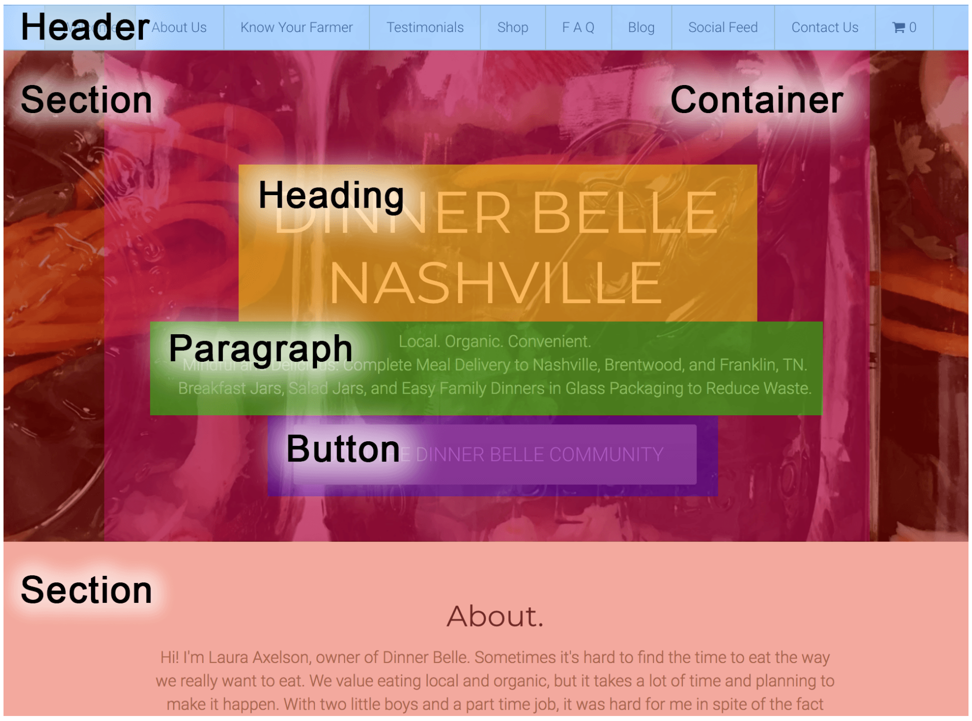
2.1.2 Website structure - Tree components

In previous chapter we presented a certain way of storing the state of a HTML markup. If we think of websites as domain components organised into a tree structure, we need to think about what kind of components are those, their possible state, their relation to the children and how they are stored.

Depending on what kind of websites we aspire to to build with our website builder, we are going to abstract the set of components for our use. The aim is to be able to build static, presentational websites that are made of one or multiple pages.

Pages share a common markup in a form of a **Layout** into which the actual markup of a **Page** is injected. A **Layout** is a template which contains components such as **Header**, **Footer** and provides a slot where a given **Page** is inserted.

Let's look at an example of a typical presentational website nowadays:



Local. Organic. Convenient.
Mindful and Delicious. Complete Meal Delivery to Nashville, Brentwood, and Franklin, TN.
Breakfast Jars, Salad Jars, and Easy Family Dinners in Glass Packaging to Reduce Waste.

Hi! I'm Laura Axelson, owner of Dinner Belle. Sometimes it's hard to find the time to eat the way we really want to eat. We value eating local and organic, but it takes a lot of time and planning to make it happen. With two little boys and a part time job, it was hard for me in spite of the fact

Figure 2.1: Example of a typical presentational website

By how content is organised in this example, we can begin to abstract the common markup starting with the most obvious patterns.

All of common presentational websites have a header, footer and page content in between them. Page content is organised into sections. Sections contain their content in a container, which has certain width and is positioned in the center of the section thus in the center of the screen. Containers then include the very vital parts of a website such as text, images, buttons and so on.

Abstracting these high level blocks we can come up with a set of components organised into a tree that holds the relationship between the components.

2.1.3 List of basic tree components

In 2.1.2 we outlined the basic building blocks of a general website we aim to be able to create with the website builder. This chapter provides a short overview of each individual component and its role in a Component tree.

- **Root**

  Root is the root component of the Component tree as it the root of website **Layout**. A website has only one layout which can be customised globally for all **Pages**.

  Whitelisted children components:

  - Header
  - Page
  - Footer

- **Header**

  Header mimics the HTML DOM `<header>` and is located on the top of the page.

  Whitelisted children components:

  - Navigation
  - Sitetitle(not implemented)

- **Navigation**

  Navigation is located inside the Header component and contains navigation items which can link to website pages or external resources.

  Whitelisted children:

  - Navigation Item

- **Footer**

  Footer mimics the HTML DOM `<footer>` and is located on the bottom of the page.

  Whitelisted children components:

  - Text
  - Image
  - Navigation
  - Button

- **Page**

  Page component serves as an abstraction of a single page of a website which is supposed to have its own unique path within the website.

  Whitelisted children components:

  - Section

- **Section**

  Section is a basic building block of a page. It generally has top and bottom padding and a background color or image. Within its container, it features content blocks such as images or texts.

  Whitelisted children components:

  - Container

- **Container**

  Container component has a certain width and is positioned in the center of a section therefore in the center of a screen. Its width adapts to the device screen size to achieve responsiveness.

  White-listed children components:

  - Image
  - Button
  - Text

- **Image**

  Image component mimics the HTML DOM `<img>` by containing an image resource users can upload.

  White-listed children components: none

- **Button**

  Button component mimics the HTML DOM `<a>` which looks like a button. It has a caption and can link to an external URL.

  White-listed children components: none

- **Text**

  Text component serves as an abstract wrapper for text content of a website. Text includes headings and paragraphs.

  White-listed children components:

  - Paragraph
  - Heading

- **Paragraph**

  Paragraph component mimics the HTML DOM `<p>` which contains text in a form of text nodes.

  White-listed children components:

  - Text node

- **Heading**

  Heading component mimics the HTML DOM `<h>`. A heading has either level one, two or three i.e `<h1>`, `<h2>`, `<h3>`.

  White-listed children components:

  - Text node

- **Text node**

  Text node mirrors the HTML DOM text node and is the basic building block of text-based components such as Heading or Paragraph.

2.1.4 Tree component implementation

Javascript classes are going to be used to represent a component inside a Component tree.

Javascript ES6 classes are syntactical sugar over JavaScript's prototype inheritance. Javascript objects in a form of instances of a class can be serialized as JSON documents. JSON can store data as Numbers, Strings, Booleans, Arrays, Objects, null value or as collections of those.

By storing only JSON's data types as attributes of a component in a Component tree, we can easily serialize the tree component implemented as Javascript class to a JSON document and send it back and forth the web server via HTTP(Hypertext transfer protocol) requests.

Let's look at Javascript class representation of an abstract tree component:

```
class Component {
  constructor(options={}) {
    this.name = options.name
    this.wbdId = uuid()
    this.settings = options.settings
    this.resources = options.resources
    this.modules = options.modules
    this.children = options.children
  }

  props() {
    return {}
  }
```

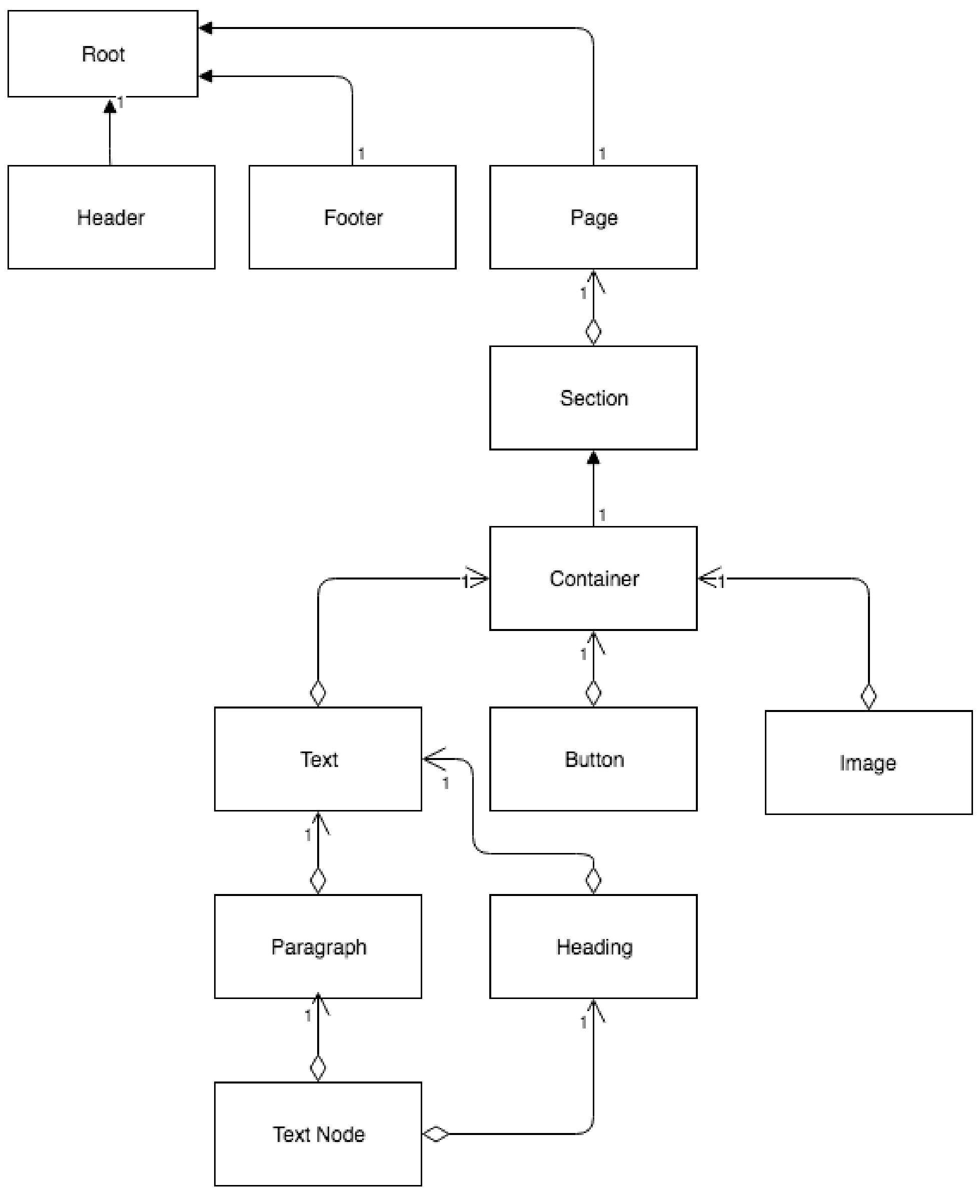


Figure 2.2: Website component tree diagram

```
  styles() {
    return {}
  }
}
```

To implement an example button class, we can inherit from a base class like this:

```
class Button extends Component {
  constructor(options={}) {
    options.name = 'wbc-button'
    super(options)
  }

  props() {
    return {
      content: this.settings.content,
      url: this.settings.url
    }
  }
}
```

Javascript ES6 classes provide us an interface to implement object-oriented concepts in Javascript environment like polymorphism or encapsulation[3].

A tree component has these attributes:

- **name**

  Name of the component

- **wbsId**

  Component unique ID evaluated after component's initialization

  wbsId is an unique identifier used to locate the component in a tree and perform modifications upon

- **settings**

  Component settings object. Properties which the component passes are evaluated based on the settings.

- **resources**

  Component resources provide an interface to link remote resource such as image or file to a component

- **modules**

  Component modules allow a component to extend its behavior

- **children**

  Component's children components

The very basic instance of a website Component tree represented as Javascript objects has this structure:

```
{
  name: 'wbc-root',
  wbcId: '1',
  children: [
    {
      name: 'wbc-header',
      wbcId: '2'
    },
    {
      name: 'wbc-page',
      wbcId: '3'
    },
    {
      name: 'wbc-footer',
      wbcId: '4'
    }
  ]
}
```

2.1.5 Modules

The concept of modules is going to be used to share common behavior between components.

A module is basically an embedded component within the parent component that has no children and can be only accessed through the component it belongs to.

Just like tree components, modules pass down properties and style to extend the behaviour or appearance of the component they belong to.

Module properties. A module has these properties:

- name
- settings
- resources

A single component can have any number of modules.

An example of a module is a **Paddingable** module which contains padding properties and can pass top and bottom padding values to its component. In our prototype, Header, Section and Footer contain the Paddingable module.

2.1.6 Modifying the Component tree

At some point, it will be necessary to modify a Component tree to change the structure or appearance of a website.

To locate any component in the Component tree, the tree will be traversed *depth-first* to find a component with matching **wbcId** property.

Updating a component

Certain components in a Component tree can be updated to alter their behavior or appearance.

It is possible to update only these attributes of a tree component:

- settings

- modules
- resources

An existing tree component is updated in these steps:

1. Get the *wbcId* of the existing component we want to update
2. Search the Component tree for the component which has matching *wbcId*
3. Update the attributes of the existing component at place

Inserting a component

Certain components can be inserted into the Component tree as children of components that have dynamic content i.e dynamic set of children.

A new tree component is inserted in these steps:

1. Get the *wbcId* of the component after or before which we want to insert a *new component*
2. Search the Component tree for the *parent component* which has a child with matching *wbcId*
3. Find the index *i* of the *parent component's* child with matching *wbcId*
4. - To insert the component after:

     Insert the *new component* as the child of the parent component at index *i + 1* at place
   - To insert the component before:

     Insert the *new component* as the child of the parent component at index *i* at place

Removing a component

Every component except the layout components in the Component tree can be removed.

An existing tree component is removed in these steps:

1. Get the *wbcId* of the component we want to remove
2. Search the Component tree for the *parent component* which has a child with matching *wbcId*
3. Find the index *i* of the *parent component's* child we want to remove
4. Remove the parent component's child at index *i* at place

2.1.7 Mapping component tree to HTML

After the component tree has been persisted, the last step of the flow is to render it to HTML so it's available as an HTML document to our users.

Each component in the component tree is assigned a template in form of a Vue.js component.

Vue.js is a Javascript framework which utilizes its components as Javascript instances that contain a HTML template and accept custom attributes in form of props. Props alter the template or behavior of a Vue.js component.

For example, **Button** tree component passes *content* and *href* to its corresponding Vue component. The template of **Button** looks like follows:

```
<template>
  <a class="wbc-button" target="_blank" :href="href">
    {{ content || 'Button text' }}
  </a>
</template>
```

Components that contain children, provide a slot in the template where their children can be rendered.

For instance, take **Container** tree component view:

```
<template>
  <div class="wbc-container">
    <slot></slot>
  </div>
</template>
```

If a **Container** contains **Image**, its instance:

```
{
  name: 'wbc-container',
  children: [
    {
      name: 'wbc-image',
      settings: { publicId: 'img.png' }
    }
  ]
}
```

is going to map to:

```
  <div class="wbc-container">
    <div class="wbc-image">
      <img src="/path/to/img.png"/>
    </div>
  </div>
```

Similarly, we can render every component in the tree like this with a mapping function *recursively*. Starting with the **Root** component, we can render the whole markup of our website and return it as HTML.

2.1.8 Persisting a Component tree

After a Component tree is retrieved from a data store, its lifecycle is typically as follows:

1. Deserialisation
2. Modification
3. Validation
4. Persistence

Let's look at the last step of the component lifecycle and assume how a Component tree can be stored in a persistent storage.

Much like HTML DOM object tree, our Component tree will form a tree of Javascript objects. However, our objects are serializable as JSON data which plays an important role in how the Component tree is going to be persisted.

There are two ways how JSON can be stored in a persistent storage as a single value:

1. Casting JSON to string, store it as a text type and subsequently parse it back to JSON after retrieval
2. Storing it as JSON type in a database system that support JSON-like types

The first option is easy to implement but makes it difficult to search or modify the tree from the data store directly.

The second option is the ideal solution although it restricts us to use a database system that supports storing JSON values.

From the popular database systems there are two options which provide this functionality:

- PostgreSQL's `jsonb` data type
- MongoDB's documents which are stored in JSON format

In this prototype, we will go with the second option as our domain logic is close to how MongoDB represents data in its documents i.e as documents and sub-documents. Additionally, for operations on deeply nested object structures, MongoDB generally provides better suited tooling.

Chapter 3
Application specification

The prototype of website builder application will be a universal web application enabling users to create, edit and view websites on the client and persist the websites in the persistent storage on the server.

Universal web applications describe JavaScript applications which run both on the client and the server.[13]

Client side contains feature-rich user interface to create websites and edit them in a WYSIWYG manner.

Server side provides API endpoints to perform CRUD(Create, read, update, delete) operations on a website and an endpoint that renders created websites.

3.1 Application requirements

This section contains functional and non-functional requirements provided by the thesis supervisor.

Functional requirements describe behavior features of the editor application.

Non-functional requirements describe qualitative features of the editor application.

3.1.1 Functional requirements

The primary use case of our application is to create a website, edit it, persist the changes and eventually view the website using the render endpoint.

A Website has a unique name and contains a Layout and Pages.

The client side application will allow users to create a website with basic Layout and initial home Page.

Client side editor will view a website in a context of the current page and enable users to edit the components through its user interface.

Functional requirements	
FR1	A user is able to create a website.
FR2	A user is able to insert a component into a layout or page.
FR3	A user is able to remove a component from a layout or page.
FR4	A user is able to change section's background color.
FR6	A user is able to align header children components horizontally.
FR7	A user is able to change text content.
FR8	A user is able to upload an tmage.
FR9	A user is able to change button text.
FR10	A user is able to add a navigation item.
FR11	A user is able to change navigation's item text.
FR12	A user is able to change navigation's item link.
FR13	A user is able to remove navigation item.
FR14	A user is able to save a website.
FR15	A user is able to view a website.
FR16	Application will show a list the created websites in a comprehensive dashboard.

Table 3.1: Functional requirements

3.1.2 Non-functional requirements

On top of functional requirements, the application must meet this quality criteria:

Non-functional requirements	
NFR1	Client application will support all ES5[14] compliant web browsers
NFR2	Part of the application will be covered by unit tests - testing tree validation logic
NFR3	Domain object model will be extensible by new tree components

Table 3.2: Non-functional requirements

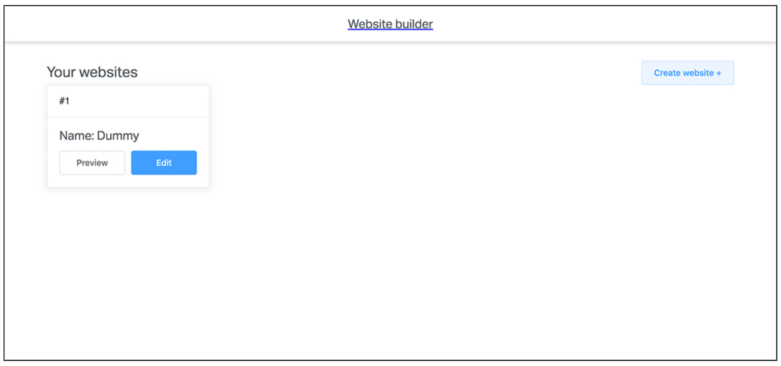
3.2 Use cases

Within the scope of our application, any user can create, edit and view websites. This section describes six use cases which feature scenarios realizing functional requirements from the application specification.

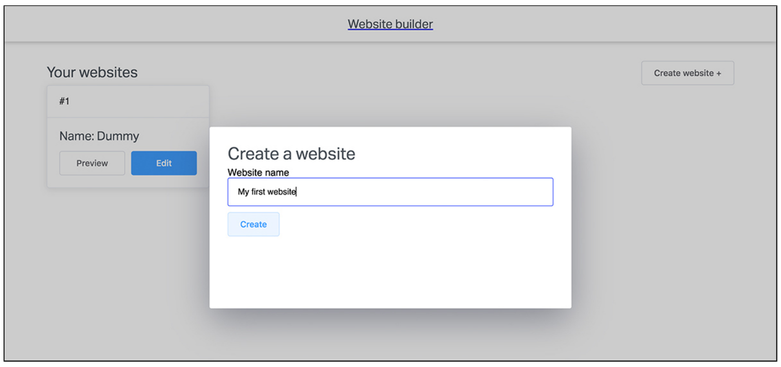
3.2.1 Create website

Users can create a website with basic **Layout** and a home **Page** from the client application 3.2.1.

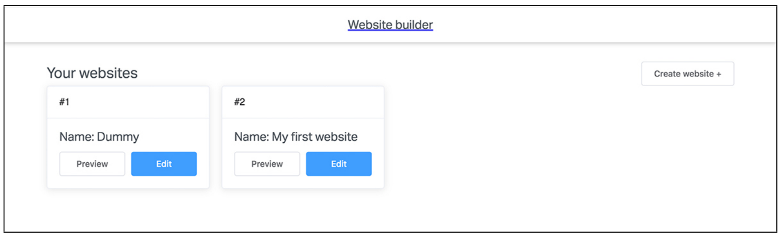
From a user standpoint, creating a website is a trivial process as it serves as a prerequisite for editing and viewing the website.



(a) : Website dashboard view



(b) : Create website form



(c) : List of created websites

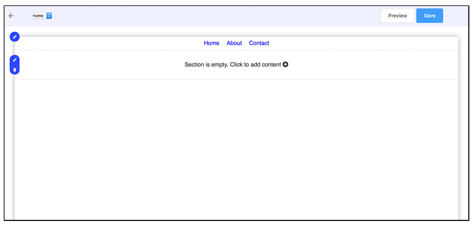
Figure 3.1: Screenshots featuring application interface - Website dashboard

Create website scenario

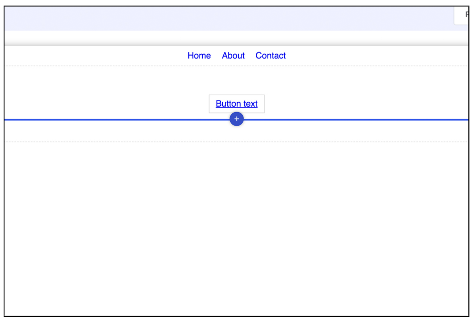
This scenario realizes these functional requirements: FR1, FR16

Scenario:

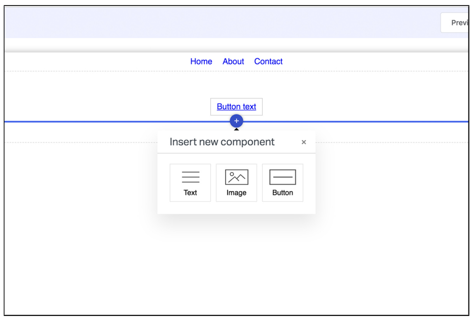
1. Users opens the Website dashboard view which contains a list of created websites and basic actions associated with them
2. User clicks the the **Create website** button
3. User interface shows the modal with website form which features a single input field - website name
4. User enters the website name and clicks on **Create** button



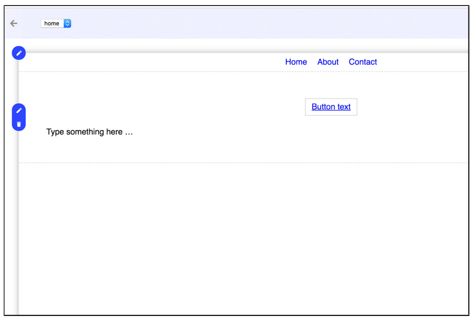
(a) : Website editor view



(b) : Tooltip that indicates a component can be inserted here



(c) : Popover menu with list of available components to insert



(d) : Website editor view with text component inserted

Figure 3.2: Screenshots featuring application interface - Inserting a text in website editor

3.2.2 Insert text into a page

A user wants to insert a new text block into a page to add some text content into a website 3.2.2.

Insert text into a page scenario

This scenario realizes these functional requirements: FR2, FR7
Preconditions:

- A website exists and contains a section where text component can be placed

Scenario:

1. User opens the Website editor view
2. User locates a section where he wants to insert new text

The section where user wants to insert a text has some content

4. User hovers the top or bottom center of a content block after or before he wants to insert new text

5. User clicks on + button that appears on top or bottom center of a content block after or before he wants to insert new text to open a popover menu with list of available components that can be inserted at place

6. User clicks on **Text** button from the popover menu

The section where user wants to insert a text has no content

4. User clicks on **Insert a new component** placeholder in the section where he wants to insert new text to open a popover menu with list of available components that can be inserted at place

5. User clicks on **Text** button from the popover menu

3.2.3 Change background color of a page section

A user wants to change the current background color of a page section to a new color 3.2.3.

Change section background color scenario

This scenario realizes these functional requirements: FR4

Preconditions:

- A website exists and contains a section which users want to change the background color of

Scenario:

1. User opens the Website editor view

2. User locates the section which background color he wants to change

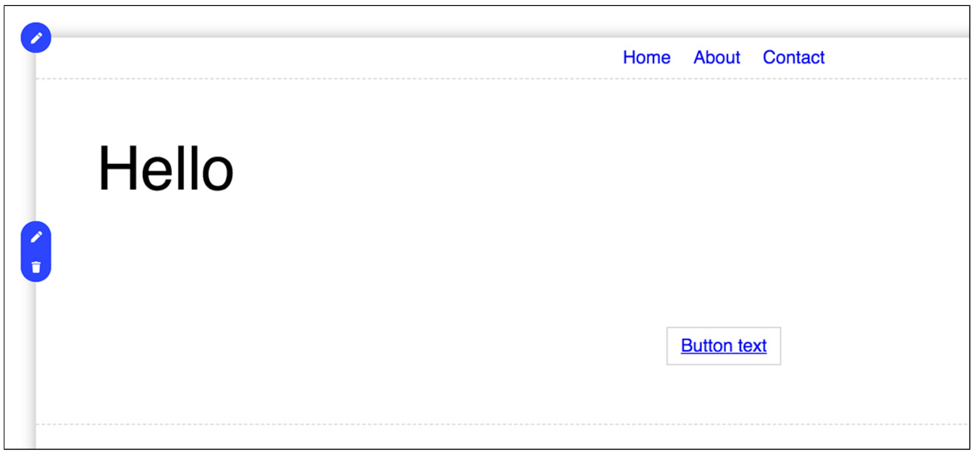
3. User clicks on **Edit icon** on the left border of the section to open the section's settings window

4. User opens the **Background tab** from the settings window

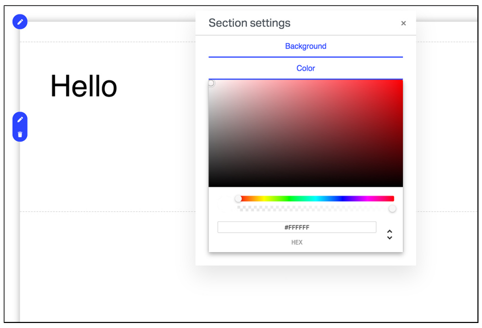
5. User picks a color from a color picker inside the **Background tab**

3.2.4 Change button text and link

A user wants to change the existing button's caption and an external URL where it links to 3.2.4.

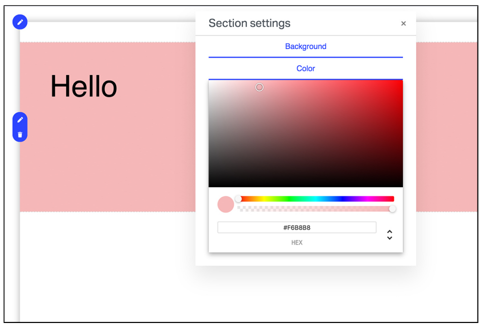


(a) : Website editor view with existing section component



#FFFFFF

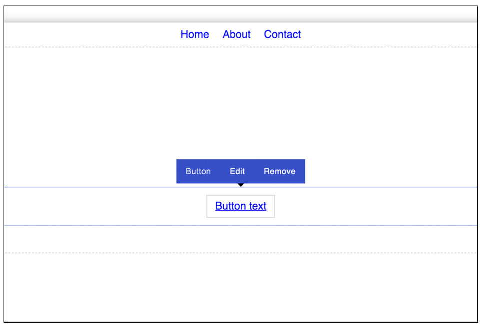
(b) : Section settings window



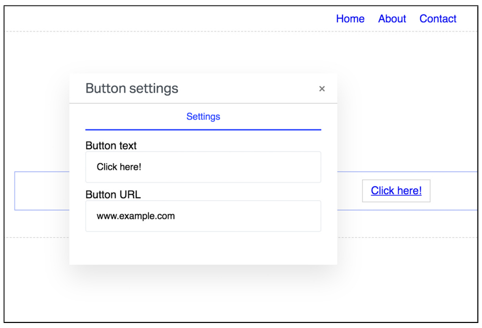
#F6B8B8

(c) : Section with changed background color

Figure 3.3: Screenshots featuring application interface - Changing section background in website editor



(a) : Button with a tooltip



Button text
Click here!
Button URL
www.example.com

(b) : Button settings

Figure 3.4: Screenshots featuring application interface - Adding a navigation item in website editor

Change button text and link scenario

This scenario realizes these functional requirements: FR9

Preconditions:

- A website exists and contains a button which users want to change the text and link of

Scenario:

1. User opens the Website editor view
2. User locates the button which he wants to edit
3. User clicks on button to show the buttons tooltips
4. User clicks on **edit icon** from the button tooltips to open button settings window
5. To change button text, user edits the text in the **Button text** text field
6. To change button link, user edits the URL in the **Button link** text field

3.2.5 Add navigation item

A user wants to add an item to a navigation located in the website header 3.2.5. Typically user wants to add a new navigation item to link to another page or an external URL

Add navigation item scenario

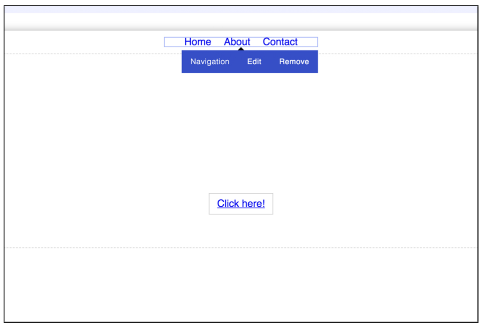
This scenario realizes these functional requirements: FR9

Preconditions:

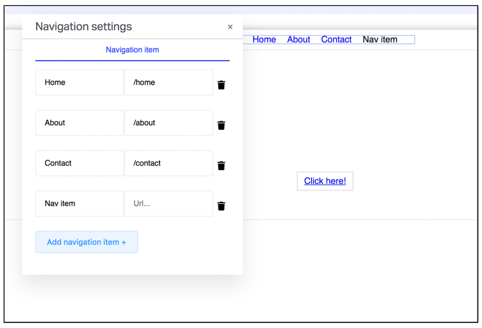
- A website exists and contains a navigation which user want to add a new item to

Scenario:

1. User opens the Website editor view
2. User locates the navigation he wants to edit
3. User clicks on the navigation to show the navigation tooltips
4. User clicks on **edit icon** from the navigation tooltips to open navigation settings window
5. To add a new navigation item, user clicks the **Add new item** from the navigation settings window. Optionally, user can change the navigation's item text at place.

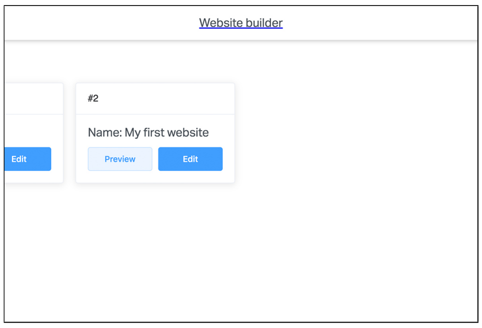


(a) : Navigation with a tooltip

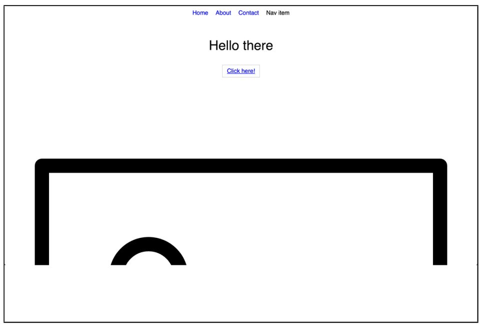


(b) : Navigation settings

Figure 3.5: Screenshots featuring application interface - Adding a navigation item in website editor



(a) : List of available websites to preview



(b) : Website preview

Figure 3.6: Screenshots featuring application interface - Viewing a website

3.2.6 View website

A user wants to view a website he created and edited with the website builder 3.2.6.

This use case demonstrates the key feature of our application which is the ability to render the websites built by our website builder on the server and return them as HTML document to the client.

Scenario:

1. User opens the Website dashboard view
2. User locates the website he wants to view
3. User clicks **View** link on the item from the list of websites

Chapter 4
Implementation

In this chapter we will describe the technology used to implement the website builder application prototype, explain the client-server side distribution and show how the website editor works under the hood.

4.1 Technology specification

4.1.1 Vue.js

Vue.js will be used to develop the user interface of the application.

Vue.js is a progressive Javascript framework designed to build modern reactive user interfaces. The interfaces consists of a collection of Vue.js components.

Components in Vue.js extend basic HTML to encapsulate reusable code. In practice, this means that markup of a component can change dynamically, depending on its state or properties.

Vue.js uses a concept of *Virtual DOM*. The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM.[6] When the state of the application changes, the Virtual DOM is re-created and updates the "real" HTML DOM only in nodes that differ. The updates to the HTML DOM are done automatically, removing the need to change view from our code manually.

Each Vue.js component has a *data* object which represents component’s state and can accept *props* which are immutable properties passed from its parent component.

Vue’s reactivity works by modifying every object added to the data object so that Vue is notified when it changes.[2]

When component state changes or props are modified, the component re-renders.

Vue.js components have this lifecycle hooks:

- **beforeCreate** - fired before component is created

  Initializes events & lifecycle

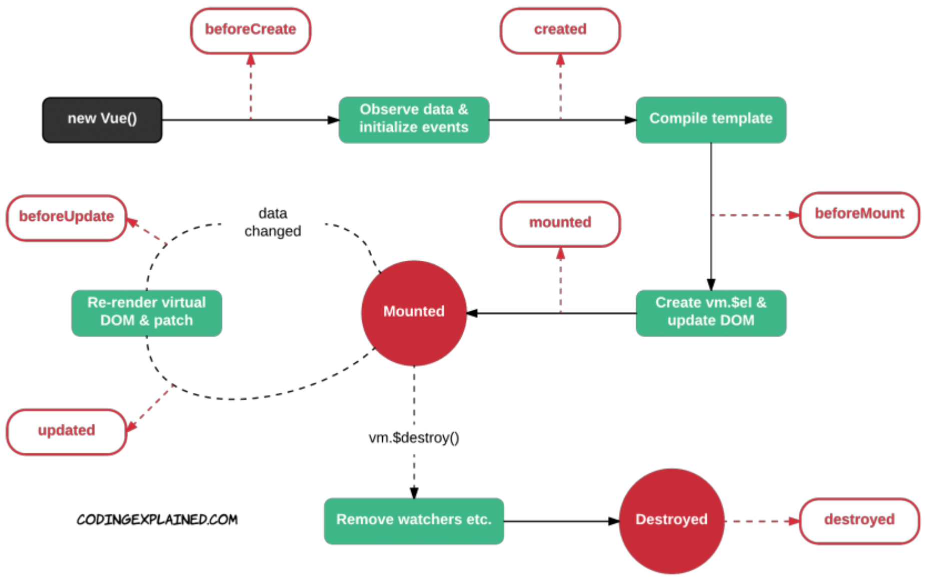


Figure 4.1: Vue.js component lifecycle [9]

- **created** - fired when component is created

  Initializes injections and reactivity
- **beforeMount** - fired when component is before mounting
- **mounted** - fired when component mounts onto HTML DOM
- **beforeUpdate** - fired when data changes and component is before update
- **updated** - fired when data changes and component updates
- **beforeDestroy** - fired when component is before being destructed
- **destroyed** - fired when component is destroyed

  Teardowns watchers, child components and event listeners

Compared to React.js, Vue.js was chosen as the application framework because its code style is similar to traditional web applications in how it separates HTML, Javascript and CSS. Vue.js component lifecycle hooks are straightforward and arguably more intuitive than in React.js. Vue.js is also more opinionated than React.js which in our case allows for quicker prototyping.

4.1.2 Nuxt.js

Nuxt.js is a framework used to create universal Vue.js web applications.

Nuxt.js will be used as the application framework of our website builder application implementing both server side and client side features.

On top of Vue.js, it provides code transpilation and bundling, automatic code splitting, routing and ultimately server side rendering which is a key feature of our website builder prototype.

Nuxt.js SSR(Server Side Rendering) lets Vue.js components to be rendered on the server, abstracting away the client-server distribution[7].

This enables us to render the websites as Vue.js applications on the server, thus define the server rendering endpoint which returns fully rendered websites as HTML documents to the client.

Server side rendering solves a common issue among modern Javascript frameworks. The initial request onto the server typically only downloads the Javascript code that initializes the application which renders the markup and mounts it on the HTML DOM.

This issue reveals a SEO(Search engine optimization) concern where HTML content of the website is not present on the first request to the server, therefore causing problems with indexing the website pages by search engines.

4.1.3 Vuex

Vuex is a state management pattern and library for Vue.js applications. Inspired by Flux pattern, developed by the Facebook company. It serves as a centralized store for Vue.js components which are running inside the application.

Complex applications face these main issues when dealing with the state:

- Multiple components depend on the same state
- A single state is being mutated from different components

Vuex solves this by extracting the application's state into a global singleton object which any Vue.js component can access or mutate.

To implement straightforward state management within our application, Vuex leverages these concepts in an instance of its store:

- **state**

  Vuex uses single state tree which acts as single source of truth for our application state

- **getters**

  Computed properties based on the store state

  A getter result is cached depending on the state of the store, re-evaluating only the the state is mutated

- **mutations**

  Mutations provide functionality to modify the state of the application

  State can be modified only from within mutations and every mutation must be a synchronous operation

- **actions**

  Actions are functions that receive the store context object and can commit mutations

  Actions are often used to fetch data from API endpoints via HTTP requests and mutate the store with the received data

[8]

In Nuxt.js, the instance of Vuex store is available in the Vue.js components by injecting it the into the components' *this* context as *this.$store*.

Vuex was chosen as the state management library for our application because it's a popular and well-documented choice among Vue.js applications. Since website and its tree components can be modified from multiple Vue.js components in a unpredictable fashion from the Editor application, Vuex will solve issue this by keeping the application state in a single centralized store. State mutations will be easily detected and it will bring more clarity to how the state is handled within our application.

4.1.4 MongoDB

MongoDB is a cross-platform document-oriented database program.[10] It is classified as a NoSQL database.

MongoDB stores data in schemaless JSON-like documents meaning their fields and structure can change over time. In the application code, MongoDB maps its documents to objects. In Node.js, these are Javascript objects.

Moreover, MongoDB is designed to be a scalable distributed database so it provides features like high availability and horizontal scaling natively.

In our application, we are going to use MongoDB to store websites as MongoDB documents. Because website **Pages** and **Layout** contain a Component tree document as their *root* attribute, MongoDB documents are ideal solution to represent and store the Component tree since it's serializable as a JSON object.

Section 5.3.1 describes how we can scale and optimise our MongoDB database.

4.2 Client application

Client application of the website builder will be implemented as a single page Vue.js application using Nuxt.js framework.

It lets users create a basic website, edit it and send it to the server application to persist it.

The client application features two views, the Website dashboard view which contains basic CRUD operations on a list of websites and the Website editor view which is the actual website builder user interface.

4.2.1 State management

The application will manage its state with a Vuex store.
It has to fetch a website, manipulate it and update it with the server.
In our client application, we're going to use two store modules:

1. **websites**

   This module stores websites loaded from the server side API and normalizes their nested structure into websites, pages and layouts.

2. **componentTree**

   This module contains a *layout* and a *page* in its state. From these two entities the componentTree module computes the Component tree which is going to be rendered and edited.

4.2.2 Communication with the server application

Client application will communicate with the application server via HTTP REST API.
It does so asynchronously using AJAX(Asynchronous Javascript and XML) requests. On the client side we will use a HTTP client library called *Axios* to implement the API calls.
Most of the communication with the server will be initiated from the application store by fetching and sending the websites from/to the server application API endpoints.

4.2.3 User interface

Client application user interface consist of two views:

- **Website dashboard view**

  The dashboard view is the home screen of the client application. It shows the list of created websites and enables users to create a website through its interface

- **Website editor view**

  The editor view lets users to edit the appearance of a website by inserting new tree components and editing or removing existing tree components.

Website editor

The Website editor view is the core part of the application as it implements the user interface of our website builder.
It displays the website in the context of the current page with a layout and lets users to edit it.
This section describes how the website editor is implemented by explaining how the website is rendered in an editable manner and how are website components edited and inserted from the editor interface.

The editor application is divided into four components with increasing stacking context:

- **Editable tree canvas**

  This component takes the Component tree object from the **componentTree** store and renders its components recursively. It does so by wrapping every component with a **wrapper**.

  Components' wrappers implement the editor features like showing tooltips and so on.

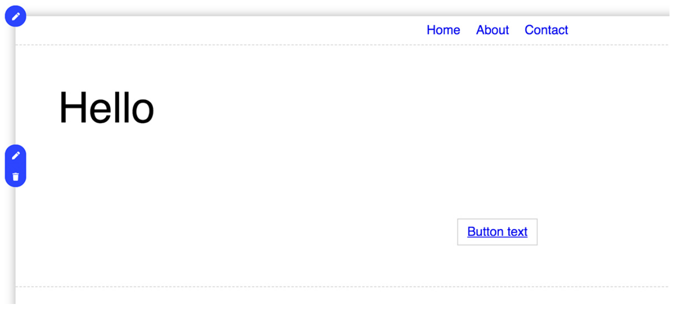


Figure 4.2: Editable tree canvas contains the rendered component tree which user can interact with

- **Component insert canvas**

  Component insert canvas filters the tree components which can have a new sibling tree component inserted before of after them. For every such component it renders an insert button which is absolutely positioned at the center bottom of a component in the editable tree canvas and shows only when user hovers around it. By clicking the insert button, a popover opens and shows available components that can be inserted into place.

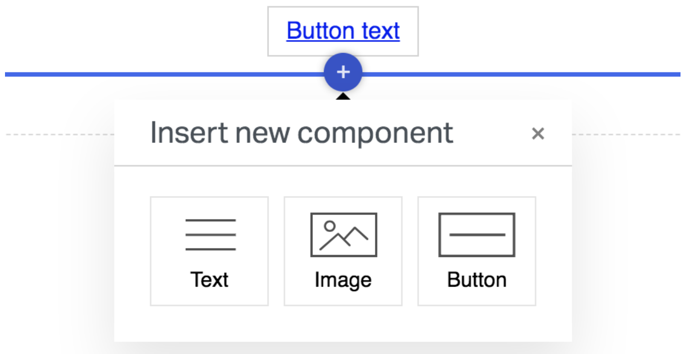


Figure 4.3: Component insert canvas contains the rendered insert tooltips that appear when user hovers around components

- **Section insert canvas**

  Section insert canvas filters the section components from the tree and renders an insert buttons which are absolutely positioned at the left corners of a section in the tree canvas. A new section is inserted at the place by clicking the section insert button.

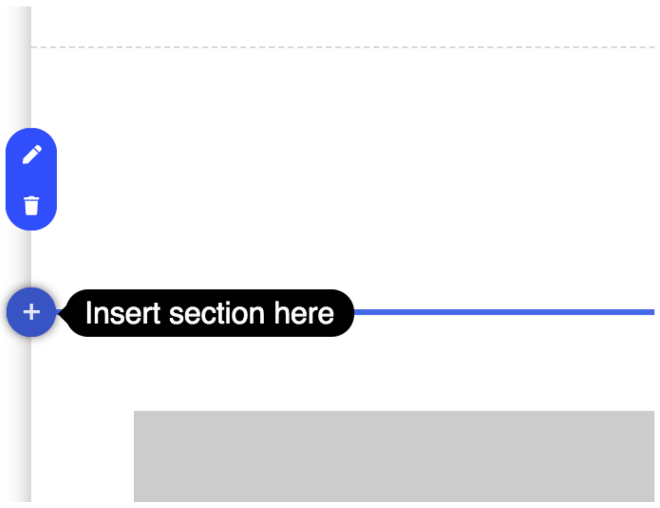


Figure 4.4: Section insert canvas contains the rendered section insert tooltips that appear when user hovers section corners

- **Component settings canvas**

  This component watches for **wbcId** of currently editing component in the store. When the wbcId of currently editing tree component is set, it shows the component's settings window that enable users to change its appearance or behavior.

4.3 Server application

4.3.1 Persistence layer

All domain data will be stored in MongoDB database.

To model our application data, we're going to use Javascript library *Mongoose*. Mongoose provides object data modeling solution that enables us to put schema on top of MongoDB documents, define validation logic and query the MongoDB collections with built-in queries.

In our application, we're going to store documents such as **Websites**, **Layouts** and **Pages**.

Let's look at the document schema of our website builder application:

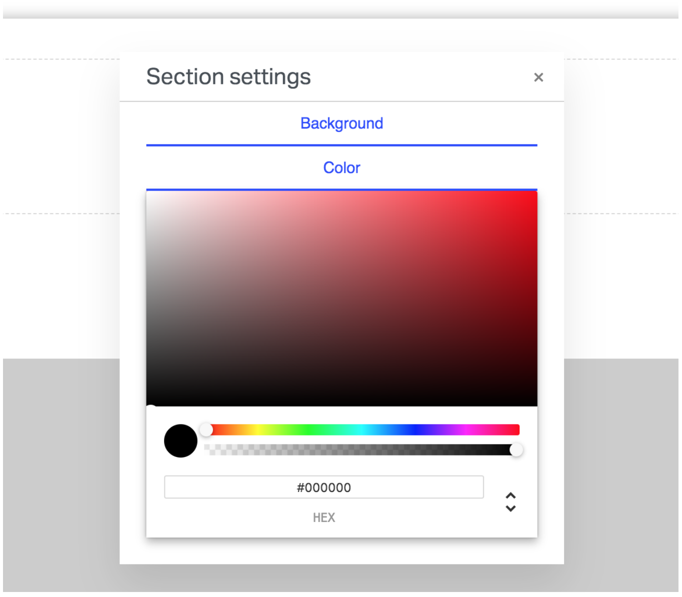


Figure 4.5: Component settings canvas renders the given component's settings window on the top of stacking context when a user is editing a component

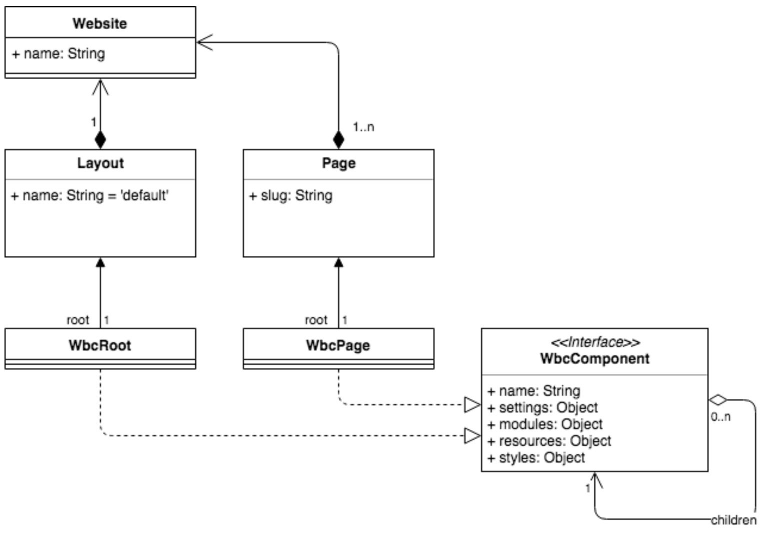


Figure 4.6: Document schema of application database

It is worth to note that MongoDB itself is a schema-less database so it might bring in some challenges when keeping the database in production environment for some time, especially when it comes to attribute name or type changes.

4.3.2 API endpoints

Editor server application provides API endpoints for website manipulation and an endpoint for rendering a given website. This section specifies these endpoints, their parameters and response.

These endpoints work with our **Website** document which has following structure:

```
{
  name: String
  layout: {
    name: String,
    root: WbcComponent
  },
  pages: [
    {
      slug: String,
      root: WbcComponent
    }
  ]
}
```

Website - create a new website

This endpoint creates a website with a given name.

POST

```
/api/websites
```

Parameters		
Field	**Type**	**Description**
name	String	Website name

Table 4.1: Create a website parameters

Success 200		
Field	**Type**	**Description**
website	Object	Website document

Table 4.2: Create a Success 200 response

Website - read website data

This endpoint returns a **Website** object

GET

```
/api/websites/:id
```

Parameters		
Field	**Type**	**Description**
id	String	Website id

Table 4.3: Read website parameters

Success 200		
Field	**Type**	**Description**
website	Object	Website document

Table 4.4: Read website Success 200 response

Website - update website data

This endpoint updates a **Website** object
PATCH

```
/api/websites/:id
```

Parameters		
Field	**Type**	**Description**
website	Object	Website document

Table 4.5: Update a website parameters

Success 200		
Field	**Type**	**Description**
website	Object	Website document

Table 4.6: Update a Success 200 response

Website - view website

This endpoint returns rendered **Website** as HTML
Get

```
/api/websites/view/:id
```

Parameters		
Field	**Type**	**Description**
id	String	Website id

Table 4.7: View website parameters

Success 200		
Field	**Type**	**Description**
website	document	Rendered website HTML document

Table 4.8: View website Success 200 response

4.4 Testing

4.4.1 Unit tests

Part of the application will be covered by unit tests.

Within the scope of this project, unit tests will be provided to test the validation of domain logic of the Component tree.

It is important to keep the domain logic in a consistent state so our websites do not break or pose a security issue within our system.

Jest will be used to implement unit tests in the application. It is a Javascript testing framework which provides unit testing functionality like matchers and mocks. Additionally, Jest is widely used in Javascript ecosystem and has a large support among both client and server side Javascript applications.

In the application we will test that our components have:

- Allowed(whitelisted) children components
- Present children components
- Unique children components
- Valid custom settings attributes

Chapter 5

Best practices

5.1 Functional components

Vue.js enables users to define so called **functional components**. A functional component is stateless and instance-less which means it has no reactive data and no *this* context.

Functional components are represented as functions therefore they are cheap to render from performance perspective.

In Vue.js, there are two ways to create a functional component:

1. Setting *functional* in the component declaration:

```
export default {
  functional: true,
  render(h) {
    ...
  }
}
```

2. Setting *functional* in the component template attribute:

```
<template functional>
  ...
</template>
```

5.2 MongoDB transactions

Unlike relational databases, MongoDB doesn't provide atomicity on multi-document transactions. This means that unless the documents are embedded in each other, writes on multiple documents are not atomic.

In case of sudden service unavailability, it might happen that some writes within a single update will pass while the others fail. In relational databases, where atomicity is one of its core features, the transaction would rollback undoing all the writes, leaving the data set in the initial state. In MongoDB, this is not a case and our domain logic may get corrupted.

However, starting in version 4.0, MongoDB adds support for multi-document transactions. This feature is only available if we configure the database to use replica sets.

It's also worth noting that generally MongoDB encourages to design a proper database schema by using sub-documents which in many cases saves users from doing multi-document transactions and only use those for special cases.

5.3 Scaling

In a situation, when our website builder hosts numerous websites and needs to render a high number of pages per second, we need to think about what scaling approaches we can take to mitigate the risk of our application underperforming.

5.3.1 Scaling MongoDB

Our MongoDB database poses as a single point of failure in our application. In case it's not available, websites cannot be rendered on the server which results in a service disruption. If the render endpoint has to handle many requests per second, database queries can slow down and increase the total response time.

Luckily, our application is not write-heavy and MongoDB can practically handle thousands of reads per second with a basic configuration.

To scale our MongoDB database, we can leverage the MongoDB's replica set.

A replica set in MongoDB is a group of `mongod` processes that maintain the same data set. Replication provides redundancy and high availability.[11] Copying the same data across multiple servers provides fault tolerance against unavailability of a single database server. In our case, we can also optimise the replica set to increase the read capacity by distributing the load to different servers.

A replica set contains several *mongod* nodes of which one is always primary node and the rest are secondary nodes. The primary node holds the write concern and confirms all write operations. Secondary nodes replicate the data in the primary by replicating primary node operation log, known as *oplog*. When the primary node becomes unavailable, the secondary nodes begin leader election as in distributed systems to choose a new primary node.

Given that our application is read-heavy, we can take these actions to scale our MongoDB database:

- Increase the size of the replica set

  Adds redundancy

- Distribute reads to secondary nodes with read preference mode

  Adds extra capacity for reads

5.3.2 Scaling application server

Our application server runs as a web server in Node.js environment.

This design decision was made intentionally since Node.js is a Javascript runtime environment and code can be easily shared across the application since both client and server side are written in Javascript.

JavaScript is single-threaded, and the way Node emulates an asynchronous environment in a single-threaded environment is via an event loop.[1] This means it never runs in multiple threads but handles concurrency by typically running blocking operations asynchronously, executing them through the event loop.

If we define our API endpoints as middleware functions of our HTTP application server, we can leverage the async nature of Node.js by implementing those middleware functions as asynchronous using Node.js's Promises. By doing this, we can achieve concurrency by having a completely asynchronous environment at very little cost.

5.4 Security

5.4.1 Protection against 3rd party code

It might be that at some point we want users to add custom code to the website. There are many use cases as in how a user can embed his own code into a website but let's examine what security issues may come up if we allow users to insert an `<iframe>` element into the website markup.

Iframes come along with several security risks:

- Cross-site scripting
- Clickjacking
- Information leak

To avoid some of these issues, we can add `sandbox` attribute to `<iframe>` element to disable certain functionality of the iframe, only allowing users to insert iframes which don't need to have certain functionality.

The `sandbox` attribute disables all these functionalities which can be enabled passed as value of the `sandbox` attribute:

- **allow-forms** - Re-enables form submission
- **allow-pointer-lock** - Re-enables APIs
- **allow-popups** - Re-enables popups
- **allow-same-origin** - Allows the iframe content to be treated as being from the same origin
- **allow-scripts** - Re-enables scripts

- **allow-top-navigation** Allows the iframe content to navigate its top-level browsing context[12]

5.4.2 TLS Protection

In case we want to host the websites built by our website builder on the application server, it is a good practice to access the websites through the HTTPS protocol. In HTTPS, the communication protocol is encrypted using Transport Layer Security (TLS)[5].

There are multiple ways how visitors could view the websites hosted with the website builder. Let's examine the three most common approaches used in multi-tenant applications:

1. Through a route on the application server such as

   `www.websitebuilder.com/view/mywebsite`

2. Through a subdomain of the domain website builder is hosted on such as `mywebsite.websitebuilder.com`

3. Through a custom domain which uses a DNS CNAME record that maps to endpoint on the application server such as `www.myswebsite.com` to `mywebsite.websitebuilder.com`

For first and second approach we could acquire a TLS certificate from certificate authority for `www` or `*` wildcard subdomain *hostnames*.

The third approach brings in complexity due to differing hostnames that can be mapped to our application endpoint. For every such hostname, the TLS certificate must be present on the server, requiring us to host and manage the certificates on our infrastructure. This is possible by acquiring the certificates dynamically from authority like *Certbot* and storing them in persistent storage or on a file system. Mind that certificates expire and it is needed to renew them periodically, bringing in more complexity to the system operation.

It is understood that it's feasible to protect the hosted websites with TLS. By using a proper approach, we can completely remove this responsibility from users to provide a layer of security without them worrying about the technical details.

Chapter 6
Conclusion

The goal of this thesis was to design and build a working prototype of a website builder which would work as a web application running in a browser.

Initially, we have pinpointed the main problems current solutions face and tried to come up with a domain logic that would model the problem we are trying to solve well.

We have set out to build a client application that would implement features like creating and modifying websites in Vue.js.

As the nature of our solution required us to somehow store oir website data that's manipulated on the client, we needed to introduce the server side logic and define how the communication with the client will look like.

The server application was implemented as a REST API powered by Node.js and we have decided to persist the data in MongoDB. The persistence layer choice was discussed in various chapters due to how important role it plays in the system.

Later it was found out that is important to keep our domain logic in a consistent state. For this purposes the unit tests were provided to ensure the integrity of our data.

Apart from the obvious challenges like user interface design or persistence layer, website builders face many other technical challenges. We have tried to cover a few of those discussing best practices in Javascript and MongoDB, scaling approaches and security concerns.

Finally, we have been able to come up with a working prototype which can be easily extended by new domain logic thanks to modern web technologies the design decisions made.

In the future, we could enrich our application by implementing features such as forms or videos or letting users to export their website code.

6.1 Final impression

Creating a website builder is a complex problem since its features and user interface directly depend on the needs of the end user and creating a website is by nature not a streamlined process.

By designing and implementing a prototype of a website builder, I greatly enriched my technical knowledge as it goes through the whole spectrum of creating a web-based product including client side development, server side development, databases, security or UX(user-experience) design.

During my bachelor studies I've deepened my engineering and web development skills with courses like *Introduction to web applications*, *Creating client applications in JavaScript* or *Enterprise architectures*.

This helped me tremendously throughout working on this thesis as I had prior understanding of the technologies and concepts used in this thesis such as client-server architecture, database systems or many more.

As website builders and web development in general are matters I want to be involved with in the future, I am very glad this thesis has served as a mean to learn and gain knowledge about these interesting and ever-evolving topics.

Bibliography

[1] Learning Node - Shelley Powers - ISBN 1449323073, 2016

[2] Vue.js: Up and Running: Building Accessible and Performant Web Apps - Callum Macrae - ISBN 1491997249

[3] Practical Object-Oriented Design in Ruby: An Agile Primer - Sandi Metz - ISBN 0321721330, 2018

[4] Wikipedia, Website builder
`https://en.wikipedia.org/wiki/Website_builder`, 2019

[5] Wikipedia, HTTPS protocol
`https://en.wikipedia.org/wiki/HTTPS`, 2019

[6] Chapter *Virtual DOM and ints Internals*
`https://reactjs.org/docs/faq-internals.html`, 2019

[7] Nuxt.js guide
`https://nuxtjs.org/guide`, 2019

[8] Vuex, State management library
`https://vuex.vuejs.org/`, 2019

[9] Vue.js instance lifecycle hooks
`https://codingexplained.com/coding/front-end/vue-js/vue-instance-lifecycle-hooks`, 2019

[10] Wikipedia, MongoDB
`https://en.wikipedia.org/wiki/MongoDB` 2019

[11] Replication in MongoDB
`https://docs.mongodb.com/manual/replication/` 2019

[12] Iframe sandbox attribute
`https://www.w3schools.com/tags/att_iframe_sandbox.asp` 2019

[13] Wikipedia, Isomorphic JavaScript
`https://en.wikipedia.org/wiki/Isomorphic_JavaScript` 2019

[14] W3Schools, ECMAScript 5
`https://www.w3schools.com/js/js_es5.asp` 2019

Appendix A

Building the application in local environment

The steps to build this application locally are following:

A.1 Prerequisites

Install following dependencies:

- Node.js >= v10.15.3
- MongoDB >= v4.0.2

A.1.1 Building the application

To build and run the application, run following commands from the project root directory:

1. `npm install -g lerna`
2. `lerna bootstrap`
3. `cd packages/wbc-admin && npm install`
4. `npm run dev`

A.1.2 Testing the application

To build and run the application, run following commands from the project root directory:

1. `cd packages/wbc-components && npm install`
2. `npm run test`