Puerro

Knowledge acquisition about how to build modern frontend web applications as simple as possible.

Semesterarbeit von

Etienne Gobeli & Robin Christen

FHNW

Hochschule für Technik

Studiengang iCompetence

Betreuender Dozent: Dierk König, Dieter Holz

Windisch, 14. June 2019

Abstract

Nowadays, most web UI's depend on frontend frameworks/libraries (like Angular, Vue or React). These technologies are powerful, but they also lock programmers in by making it difficult to have a presentation layer independent of the rendering technology.

Each technology has their own *way of working* which doesn't integrate well with others. Therefore, learning a new framework/library could be a bad investment, as the technology might be outdated soon.

With this project, we want to research and evaluate different approaches to build modern frontend web applications as simple as possible. In order to not lose our train of thoughts, we limit this project's scope to frontend development using the technology standards HTML5, CSS3 and ES6.

It turned out, that there is not one single best way to handle every problem. Depending on the size and complexity of the application, different approaches should be considered.

If the goal is a dynamic informational website or a small web application, without a lot of internal state and little changing parts, the easiest solution is using plain and direct DOM manipulations with JavaScript.

As soon as there is a lot of changing content, direct DOM manipulations can get chunky and hard to maintain. Therefore, it is advisable to use a virtual DOM in order to change multiple elements regularly. A virtual DOM abstraction is useful for single view applications with huge and periodically changing DOM trees.

When developing frontend applications, handling and storing data on the client becomes quickly relevant. Normally, when state changes, the DOM needs to be updated accordingly. Doing this manually could lead to redundant code and doesn't comply with the separation of concerns pattern. Among other options, there is the possibility to combine state management with the virtual DOM and delegating the DOM access to a centralized location.

Another approach to separate the data, view and logic is with the well-known MVC pattern. Using its unidirectionally dataflow has the benefit of an improved understandability and maintainability especially for larger applications with a lot of logic. The only abstraction needed for this to work are observables.

For most applications, using a framework is not necessary. It comes with many dependencies, more features you will ever need, and it is hard to fully understand every peculiarity. Plus, it locks a developer in a predefined structure which makes it hard to switch the technology once started. Therefore, it is often better to reach for simplicity instead of predetermined complexity.

Table of Contents

1 Introduction

The Puerro project is a knowledge acquisition about how to build modern frontend web applications as simple as possible by researching different approaches. It can be used as a knowledge base or in combination with the provided abstractions.

1.1 About Web Development

Coding for the web means that the end-user can access the result through a browser (e.g.: Chrome, Firefox, Safari, Edge or Internet Explorer).

A browser can request data from different sources (servers) and display the received content. This content is usually a combination between these 3 different programming languages:

- **HTML**: Describes the content and structure of the page
- **CSS**: Describes the presentation of the page
- **JavaScript**: Adds behaviour to the page - and makes it more dynamic

A website is **static** when its content is fixed. It only uses HTML/CSS and there is no way to interact with it (other than linking to other pages). If you print it out, it still works.

A website is **dynamic** when its content or structure changes. This can be achieved with a server-side (backend) or client-side (frontend) approach. When the content changes from being just informational to being more interactive, we say that it is a **web application**.

With a **Server-Side** approach, the logic runs on the server, modifies the web page and sends the result back to the client. There are several programming languages available to work with it, like *PHP*, *Ruby*, *Java* etc.

With a **Client-Side** approach, a script runs in the browser of the user. This is being achieved with the delivered **JavaScript**.

Both approaches have their advantages and disadvantages.

One disadvantage of the server-side approach is, that with every request the browser needs to reload the whole page. This is very time consuming and doesn't provide a great user experience.

Amongst other reasons, that is why the modern way to develop web applications is with client-side scripting.

Client-side scripting allows for remote API calls (also known as **Ajax** calls). With these the browser can asynchronously load only the data needed and dynamically change the web page with JavaScript. This provides a much better user experience as the page doesn't need to reload completely.

1.2 Purpose

Nowadays, most web UI's depend on front-end frameworks/libraries (like Angular, Vue or React). These technologies are powerful, but they also lock programmers in by making it difficult to have a presentation layer independent of the rendering technology.

Each technology has their own *way of working* which doesn't integrate well with others. This makes it, especially in the fast changing web front-end world, a challenging problem.

It takes a lot of time to learn a new framework/library and to build an application with it. This could be a bad investment, as the technology is very likely outdated soon, and the application would need to be rebuild nearly from scratch.

Additionally, they usually build up a huge dependency chain. This means that you, as the end-developer, can't control every part of your application. If only one of the dependencies is insecurely built, fails, or runs something which it is not supposed to, it puts your entire application at risk.

The goal of this project is to explore new, unconventional, and unorthodox approaches for frontend web development with JavaScript and validate their usefulness against typical UI patterns.

1.3 Methodology

The purpose of this project is to research and evaluate frontend approaches which can be done in different ways. One of the main concerns is, that the different ideas are not only looked at on an abstract level far from reality. That is why the main research for this project is done as a project simulation. This simulation is separated into iterations with ever increasing requirements for a fictional web application.

With each iteration the imaginary customer communicates requirements to the developer which is implementing them. With the specification in place, multiple approaches are evaluated, and one is chosen to implement the customers wishes.

The claims made during this project documentation would lose their significance with a project that is too specific. That is why the outcome of the fictional project is essentially a CRUD (Create, Read, Update, Delete) application with different views. Implementing CRUD's is also one of the main purposes of many of the modern frontend frameworks and generally a good use case to test the hypotheses made during the project.

1.4 Scope

To make our examples and research work in real-life scenarios there would be a need for backend systems to store data and handle business logic. The scope of this project, however, is limited to client-side scripting. This restriction is put in place to ensure, the project does not scratch the surface of different parts of web development, but to dive deep into the specific world of frontend development.

To explore how JavaScript and HTML work and interact with each other in depth, styling, specifically CSS is also considered out of scope.

Build tooling and browser compatibility is a big part in today's frontend world. Again, to keep the focus and not get distracted by tooling, this project is respecting the following standards and the browsers which support those:

- HTML5
- CSS3
- ES6 (JavaScript version)

1.5 Coding Conventions

1.5.1 Testability

Since the code for the web behaves differently for each browser and version, it is especially important to write test cases. At the same time, the technologies are changing constantly in the web community. Therefore, we need a way to always make sure that the code is running as it is supposed to.

There are many different types of testing. We want to use unit testing and functional (frontend) testing.

- Unit Testing: To test the created abstractions (developer-view)
- Functional (frontend-only) Testing: To test the Huerto iterations for its functionality (customer-view)

For the actual testing, there are many different tools available to support the process. These are powerful but also complicated. That is why we decided to write our own little testing utility.

Basically, we just need a way to compare two values for equality, report the result and give some detailed information in case of failure. Most of our tests will run in the browser and are using the DOM. With this testing utility, we are using the executional context of the tests simultaneously for the final report.

This also brings the advantage of being able to run the tests with a desired browser and check if it still works.

1.5.2 Bundling

In JavaScript, the code to be executed needs to be loaded into the same namespace. This can create name clashes and comes with some safety issues, as suddenly external code can run a module's internal methods.

Until ES5 this problem has been handled with the *Revealing Module Pattern* which uses IIFE's and its function scope to hide the internal variables and methods. Since ES6, JavaScript supports modules which can be exported and imported.

In this project we presuppose an ES6 environment and use the ES6 modules.

These modules, however, are loaded using the *Same-Origin* policy. This basically means that a web server is needed to serve the files. To develop locally, we would need to run a local web server or use a bundler.

- The local web server would add the needed headers to comply with the *Same-Origin* policy but could create cashing problems.
- The bundler would create IIFE's out of the modules and put it all together in one file. With this approach name clashes could appear.

We decided to use the bundler *rollupjs* to allow us to develop locally without the use of a web server.

We also want to provide the easiest possible way to interact with our findings. Therefore, we decided to go against the general recommendation of not checking in generated artefacts. This brings the advantage that our GitHub project can be cloned and used, without the need to generate the bundles first.

1.5.3 JSDoc

To provide a better code understanding, we use the markup language JSDoc to annotate our JavaScript files. Using JSDoc comments, we describe the interface and usage of our functions. As with Javadoc, JSDoc can be used to generate a documentation in accessible formats like HTML. Another powerful feature of JSDoc is that in in modern text editors (e.g. VS Code or WebStorm) the type of the parameters can be taken into consideration. This allows to use a better intellisense and provides a way for checking types.

1.5.4 Variables Notation

To better differentiate between our variables, we prepend them with the following prefixes:

- `$` - when referring to real DOM's, e.g. `$div`
- `v` - when referring to virtual DOM's, e.g. `vDiv`

In case variables are unused and can be ignored, we will simply use an underscore to signalize that it is *throwawayable*.

```
$element.addEventListener('click', _ => console.log('clicked'));
```

2 DOM

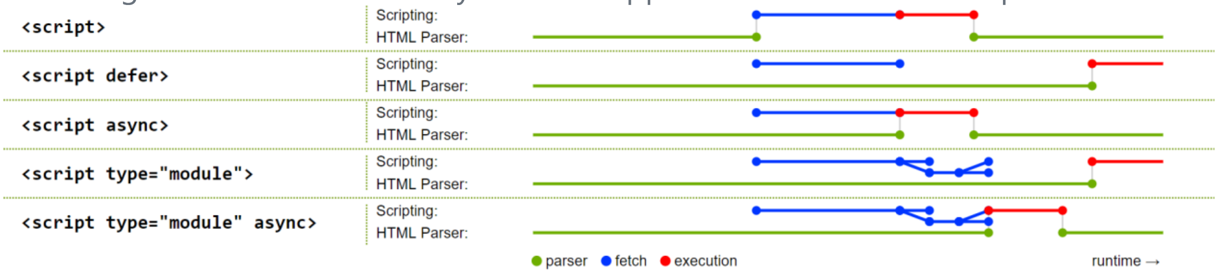
While a browser receives an HTML source file, it starts to parse it piece by piece to construct an object representation of the document called the DOM (Document Object Model) tree. Simultaneously, the DOM is being converted into a *render tree*, which represents what eventually is being painted. The document can start being rendered and painted in the browser before it is fully loaded. Unless it is blocked by CSS or JavaScript.

When the parser comes across CSS code, the rendering process is being blocked until the CSS is fully parsed. Similar as before, the browser now constructs a CSSOM (CSS Object Model) tree which associates the styles to each node. After parsing the CSS, a combination of the DOM and CSSOM are being used for continuing creating the *render tree*.

Even though the rendering process can be blocked by CSS, the DOM is still being parsed. Unless it comes across JavaScript. When the parser reaches a `<script>` tag, the parsing stops, and the script is being executed. This is the reason that a JavaScript file needs to be placed after the appearance of a referenced element in the script. Previously, it has been best practice to always include the `<script>` tags at the end of the document to make sure that all the elements are available and to not block the rendering and therefore painting process of the browser.
Nowadays, there are other options to avoid that JavaScript is being parser blocking. For example:

```
<script async src="script.js"> // Script is executed asynchronously, while the
page continues parsing
<script defer src="script.js"> // Script is executed when the page has finished
parsing
```

This image summarizes the many different approaches to load JavaScript code.



More information can be found in the HTML5 specification

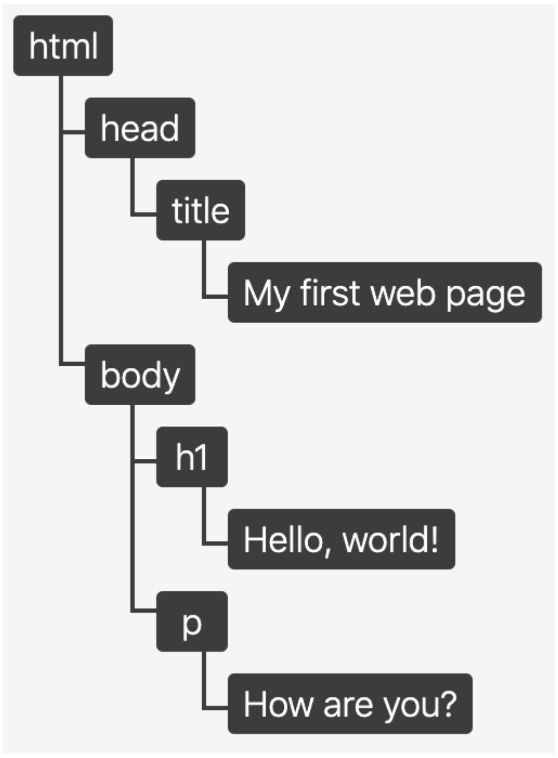
2.1 Manipulating the DOM

Basically, the DOM is an interface for HTML (and XML) documents which represents the page. It is dynamic, and the browsers provide an API to read and change the content, structure, and style of the document via JavaScript. This allows for changing parts of the website without the need of a refresh and therefore a repaint of the whole page.

JavaScript has access to a global object. In a browser, `window` is the global object and represents the window/tab of the browser in which the script is running. One of its property is `window.document` which serves as an entry point to the parsed DOM tree. Because `window` is the global object, there is no need to reference its properties (e.g. `document`) via `window`. The property name can be used directly as the script will figure out the global object at runtime.
The `document` interface can now be used to manipulate the DOM.
As an example let's create an HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <title>My first web page</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
    <p>How are you?</p>
  </body>
</html>
```

The browser will parse this file to the following DOM:



This DOM can now be manipulated via the `document` interface in JavaScript:

```
const $h1 = document.querySelector('h1'); // Accessing the <h1> element
console.log($h1.textContent);             // Reading: Hello, world!
$h1.textContent = 'Hello, Puerro!';       // Manipulating

const $button = document.createElement('button'); // Creating new Element
document.body.appendChild($button);               // Changing DOM structure
```

2.1.1 TextContent vs. InnerText vs. InnerHTML

There are three different ways to access the content of a DOM element.

1. **`element.textContent`**: represents the text content of a node as it is in the DOM. Therefore, it doesn't include the HTML tags but keeps the content of non-visible elements. For example, the content of `<script>` or `<style>` tags.
2. **`element.innerText`**: similar to `textContent` but uses CSS knowledge and only returns visible content. This has the disadvantages that reading a value with `innerText`triggers a reflow to ensure up-to-date computed styles. This can be computationally expensive and should be avoided when possible.
3. **`element.innerHTML`**: represents the HTML source of the element. It should only be used when the intention is to work with HTML markup. Misusing it for text is not optimal for performance and it is vulnerable to XSS attacks.

2.1.2 Creating Elements

The `element.innerHTML` method allows to build up a nested structure using the HTML markup language relatively easy.

```
document.body.innerHTML = `
  <input type="number" />
  <button type="button" onclick="console.log('Clicked')">Do it!</button>
`;
```

This gets slightly more challenging for appending or modifying nodes to an already existing DOM as all its child elements are being re-parsed and recreated completely. This means that saved references to nodes are no longer pointing to the supposed elements.

Re-parsing the whole structure of the element is also bad for the performance.

```
document.body.innerHTML = '<h1>Tomato</h1>';
const $h1 = document.querySelector('h1');
document.body.innerHTML += '<p>Lean as a Leek</p>'; // Whole body element is being
reparsed
$h1.textContent = 'Puerro'; // Reference does not point to the DOM instance
```

There is a solution for this called `element.insertAdjacentHTML` which does not re-parse all its child elements.

```
document.body.innerHTML = '<h1>Tomato</h1>';
const $h1 = document.querySelector('h1');
document.body.insertAdjacentHTML('beforeend', '<p>Lean as a Leek</p>'); // No
complete reparsing
$h1.textContent = 'Puerro'; // Reference still works
```

This combination between `innerHTML` and references is not very readable. Plus, when dealing with registering event listeners as well, it can get complicated. If a reference to a created element should exist at a later time, it is advisable to use the `document.createElement` method.

```
const $input = document.createElement('input');
$input.setAttribute('type', 'number');
$input.setAttribute('value', 1);

const $button = document.createElement('button');
$button.setAttribute('type', 'button');
$button.textContent = 'Go';
$button.addEventListener('click', _ => console.log($input.value));

document.body.append($input, $button);
```

Puerro provides an abstraction to make it more convenient to create elements.

```
const $input = createDomElement('input', { type: 'number', value: 1 });
const $button = createDomElement('button', { type: 'button', click: _ =>
console.log($input.value) }, 'Go');
document.body.append($input, $button);
```

2.2 Event-Driven Actions

After the HTML has been parsed, rendered and painted, the browser is usually waiting for user interactions. For this to work the browser uses an event-driven programming model to notify the JavaScript code about what's happening on the page.

There are a lot of different events. For example, when the DOM is finished with loading, clicking elements, typing on the keyboard, scrolling and many more.

In order react to an event, `Event Handlers` are used. Event handlers are functions which are being called from the browser when an event occurs.
When an event is fired, the first parameter an handler receives is an `Event` object which contains useful information and methods. The most used are:

- `target`: A reference to the target to which the event was originally dispatched.
- `type`: The name of the event.
- `stopPropagation()`: Stops the propagation of events further along in the DOM.
- `preventDefault()`: Cancels the event.

2.2.1 Register Event Handler

To register an event there are three possibilities.

2.2.1.1 Inline Event Handlers

The most legacy but direct way is to register event handlers directly in the HTML markup.

```
<button onclick="console.log('Button clicked');"></button>
```

But especially for larger projects this is considered a bad practice as it is hard to read and maintain because it doesn't separate the view from the actions. It also requires that the functions are exposed globally, which is pollution to the global namespace.

2.2.1.2 DOM on-Event Handlers

A better way is to register the event handlers in the JavaScript code. It is similar to the inline event handlers, but it respects the separation of concerns and the scope is more controllable.

```
$button.onclick = event => console.log('Button clicked');
```

A drawback with this approach is that it is only possible to assign one listener to each event.

2.2.1.3 Using `addEventListener()`

The most modern approach is to use the `element.addEventListener()` function. It allows to register as many event handlers as needed.

```
$button.addEventListener('click', event => console.log('Button clicked'));
```

With this approach it is also possible to remove listeners with the `element.removeEventListener()` function. Another advantage is the ability to choose between event bubbling and capturing.

2.2.2 Bubbling and Capturing

When nodes are nested, a user interaction can trigger multiple events. Two different models exist to handle this:

- **Bubbling** (default): The event propagates from the clicked item up to all its parents, starting from the nearest one.
- **Capturing**: The outer event handlers are fired before the more specific handler.

With the following example, the events bubble. Meaning they are propagated upwards.

```
<body>
  <div>
    <button>Click Me!</button>
  </div>

  <script>
    const $div    = document.querySelector('div');
    const $button = document.querySelector('button');

    $div   .addEventListener('click', _ => console.log('DIV clicked'));
    $button.addEventListener('click', _ => console.log('BUTTON clicked'));

    // Console Output:
    // BUTTON clicked
    // DIV clicked
  </script>
</body>
```

To make sure that the `DIV` event listeners triggers first, the methods `useCapture` parameter needs to be true. All the event handlers with `useCapture` enabled run first (top down), afterwards the bubbling handlers (bottom up).

```
$div.addEventListener('click', _ => console.log('DIV clicked'), true);
```

To complete stop the propagation, the handler can call the `stopPropagation()` method on the event object.

```
$button.addEventListener('click', event => {
  console.log('BUTTON clicked');
  event.stopPropagation();
});
```

2.2.3 Forms

Building forms is a widely used pattern for web applications. HTML provides a `<form>` tag, which allows to group interactive controls together for submitting data to a server. When a form is being submitted, an HTTP Request with the specified method is sent to the specified resource. With this traditional approach, the page always will be refreshed and new rendered based on the response.

This approach is acceptable if we want to display a completely different view after submitting the form. However, for modern web application this is usually not desirable. Instead, it is better to use an Ajax request in the background without affecting the page and to manipulate the DOM based on the response.

Nevertheless, using the `<form>` tag has many advantages and should still be used for grouping interactive controls:

- It improves the logical structure of the HTML.
- It increases the accessibility for screen readers.
- It provides a better user experience on mobile phones.
- It has the ability to access its elements conveniently.
- It has the ability to easily reset its elements.

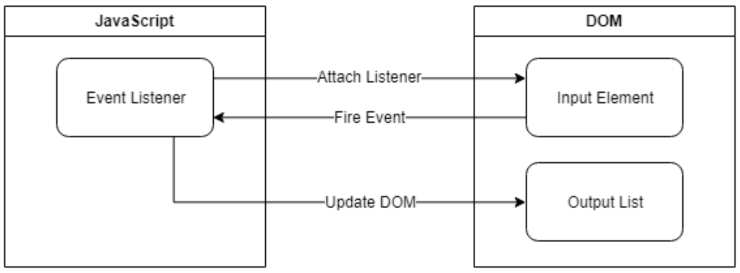
In order to use the `<form>` tag without it being submitted, an event handler has to be registered for the form's submit event. In this handler the method `event.preventDefault()` can be executed to prevent the form from submitting.

```
<body>
  <form>
    <input name="name" />
    <input name="age" type="number" />
    <button>Submit</button>              <!-- Submits Form -->
    <button type="reset">Reset</button> <!-- Resets Form  -->
  </form>
  <script>
    const $form = document.querySelector('form');
    $form.onsubmit = event => {
    // Proccess form elements at will
    // (e.g. Ajax Request, Validation, DOM manipulation)
      console.log(event.target.name.value); // Easy access on name value
      console.log(event.target.age.value);  // Easy access on age value
      event.preventDefault();               // Prevent form submitting
    };
  </script>
</body>
```

A button can have 3 different types: `submit`, `reset` and `button`. The default type is `submit` which will attempt to submit form data when clicked. When the intention is to use a button without a default behaviour, explicitly specify `type="button"`.
A `<form>` can also be submitted by pressing *enter* or via JavaScript. Therefore, using a `<button type="button">` with a click event handler won't be enough. Plus receiving the target form in the event is a huge benefit.

2.3 Testability

When event handler functions receive events, they can in turn manipulate the DOM.



For a handler function to manipulate the DOM, references to the elements which have to be manipulated are needed. Those references can either be already available in the surrounding scope or can be created in the function itself.

```
const handleEvent = event => {
  const $element = document.querySelector('div');
  // manipulate $element
};

$element.addEventListener('click', handleEvent);
```

With the help of eta reduction, the parameter can be removed when there is only one argument or when using curried functions. `x => foo(x)` can be shortened to `foo`

This gets problematic when the intention is to test this unit, since the DOM might not be available. Furthermore, this approach can quickly become difficult to maintain.

A better approach for a simplified testability is to receive the nodes which are being manipulated as a parameter.

The element which fires the event does not have to be passed as an argument because it is available through `event.target`.
When new elements are being created, it is a good practice to return them for a more convenient testing.

```
import { createDomElement } from 'puerro';

export { appendInput, changeLabel };

const appendInput = ($input, $output) => _ => {
  const $element = createDomElement('p', {}, $input.value);
  $output.append($element);
  return $element; // return for testing purposes
};

const changeLabel = $button => event => {
  $button.textContent = 'Save: ' + event.target.value;
};
```

To use the handler functions, they simply have to be registered with the needed references as arguments.

```
import { changeLabel, appendInput } from './example';

const $input  = document.querySelector('input');
const $button = document.querySelector('button');
const $output = document.querySelector('output');

$button.addEventListener('click', appendInput($input, $output));
$input .addEventListener('input', changeLabel($button));
```

To use the handler functions for testing, the needed elements need to be created.

```
import { describe, createDomElement } from 'puerro';
import { appendInput, changeLabel } from './example';

describe('Testable Units', test => {
  test('appendInput', assert => {
    // given
    const $input  = createDomElement('input', { value: 'Puerro' });
    const $output = createDomElement('div');

    // when
    const $element = appendInput($input, $output)(); // event object not needed

    // then
    assert.is($output.children.length, 1);
    assert.is($element.tagName, 'P');
    assert.is($element.textContent, 'Puerro');
  });

  test('changeLabel', assert => {
    // given
    const $input  = createDomElement('input',  { value: 'Puerro' });
    const $button = createDomElement('button', { type: 'button' });

    // when
    changeLabel($button)({ target: $input }); // mocking event object

    // then
    assert.is($button.textContent, 'Save: Puerro');
  });
});
```

The above example can be found in the Puerro Examples.

2.4 Use Cases

This style of programming with direct DOM manipulations within the event handler functions is easy and intuitive. It can be used for various tasks:

- Dynamic informational websites.
- Reactive content without side effects.
- Experimenting/Prototyping.
- Simple Web application without many changing parts.
- Server-Side rendered web applications.

2.4.1 Advantages

- Zero dependencies.
- Little code.
- Easy to understand.
- Fast.

2.5 Problems / Restrictions

This approach is getting harder to maintain when either frontend state is being introduced or there are many changing elements. This is especially true when the application starts growing.

When an event triggers a lot of changes, a reference to each dependent element needs to be managed. If multiple elements need to be updated constantly and in short time frames, it can start to become expensive to constantly query and update the DOM. Furthermore, when updates depend on data stored in the DOM, the decapsulation between view and model is not given.

For large application there probably will be redundant code and all DOM related accesses are scattered through the code.

2.5.1 Disadvantages

- Difficult to scale and maintain.
- Hard to structure/organize.
- Redundant code.
- Scattered DOM manipulations.
- Very specific.
- State lives in the view.

3 Virtual DOM

When the DOM was invented in 1998, websites were built and managed differently. It was not common to regularly modify the structure of the page via the DOM API. Constantly updating multiple elements on a page can get very chunky, hard to maintain and even performance intensive.

This is a simple example to change a table item and adding a new row:

```
<body>
  <table>
    <tbody>
      <tr class="row">
        <td class="item">Tomato</td>
      </tr>
    </tbody>
  </table>

  <script>
    const $tbody = document.querySelector('tbody');
    const $td = $tbody.querySelector('td');
    $td.textContent = 'Puerro';

    const $newTr = document.createElement('TR');
    $newTr.setAttribute('class', 'row');

    const $newTd = document.createElement('TD');
    $newTd.setAttribute('class', 'item');
    $newTd.textContent = 'Huerto';

    $newTr.append($newTd);
    $tbody.append($newTr);
  </script>
</body>
```

Most of the time it is easier to perform more expensive operations and updating larger parts of the DOM. This includes the re-rendering of nodes which don't change. As seen in the last chapter, this has some disadvantages, as references and registered event listeners are getting lost. Plus, it can be vulnerable for XSS attacks if the content is not being sanitized first.

```
const $tbody = document.querySelector('tbody');
$tbody.innerHTML = `
  <tr class="row">
    <td class="item">Puerro</td>
  </tr>
  <tr class="row">
    <td class="item">Huerto</td>
  </tr>
`;
```

The virtual DOM solves the problem of needing to frequently update the DOM. It is not an official specification, but rather a new method of interfacing with the DOM. It can be thought of as an abstraction or copy of the DOM.

Basically, the virtual DOM is just a regular JavaScript object representing HTML markup. It can be manipulated freely and frequently without using the DOM API. Whenever needed, it can execute the specific changes it needs to make to the original DOM.

Puerro has its own implementation of the virtual DOM. Check it out.

3.1 Creating a Virtual DOM

Since a virtual DOM element is just a JavaScript object, which can be created like this:

```
const vDOM = {
  tagName: "tbody",
  attributes: {},
  children: [
    {
      tagName: "tr",
      attributes: { class: "row" },
      children: [
        { tagName: "td", attributes: { class: "item" }, children: ["Puerro"] }
      ]
    },
    {
      tagName: "tr",
      attributes: { class: "row" },
      children: [
        { tagName: "td", attributes: { class: "item" }, children: ["Huerto"] }
      ]
    }
  ]
};
```

Using Puerro simplifies this a lot:

```
const vDOM = h('tbody', {},
  h('tr', { class: 'row' }, h('td', { class: 'item' }, 'Puerro')),
  h('tr', { class: 'row' }, h('td', { class: 'item' }, 'Huerto'))
);
```

`h` stands for *hyperscript* and is a common abbreviation for building virtual elements.

Since the virtual DOM will be modified frequently, it is a good practice to have a function to create the virtual DOM with the changing parts as parameters.

```
const createVDOM = items => h('tbody', {},
  items.map(item =>  h('tr', { class: 'row' }, h('td', { class: 'item' }, item)))
);
```

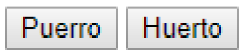
3.1.1 White-Spaces

Spaces, tabs, or line breaks are all white spaces which are used to format code. In HTML markup, these white spaces are normally only for readability purposes and are not impacting the layout of a page.

However, there are exceptions. In case there are white spaces between inline elements, they are getting collapsed and displayed as a single space.

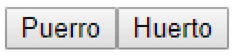
```
<button>Puerro</button>
<button>Huerto</button>
```

The newline will be represented and displayed as a text node with a space in the final layout.



Without the white spaces, there is no gap between the elements.

```
<button>Puerro</button><button>Huerto</button>
```



How exactly white spaces are handled can be read in the [CSS Text Module Level 3 Specification]().

This needs to be taken into consideration when creating virtual elements since white spaces are not being created. A possible workaround could be to use an advanced templating language like JSX.

3.1.1.1 JSX (JavaScript XML)

JSX is a pre-processor step which allows to write virtual DOM elements with the HTML/XML syntax. Basically, it compiles XML syntax into the above described notation using the `h` function before using it as a JavaScript file.
This project does not go into more detail on how it works and how to use it.

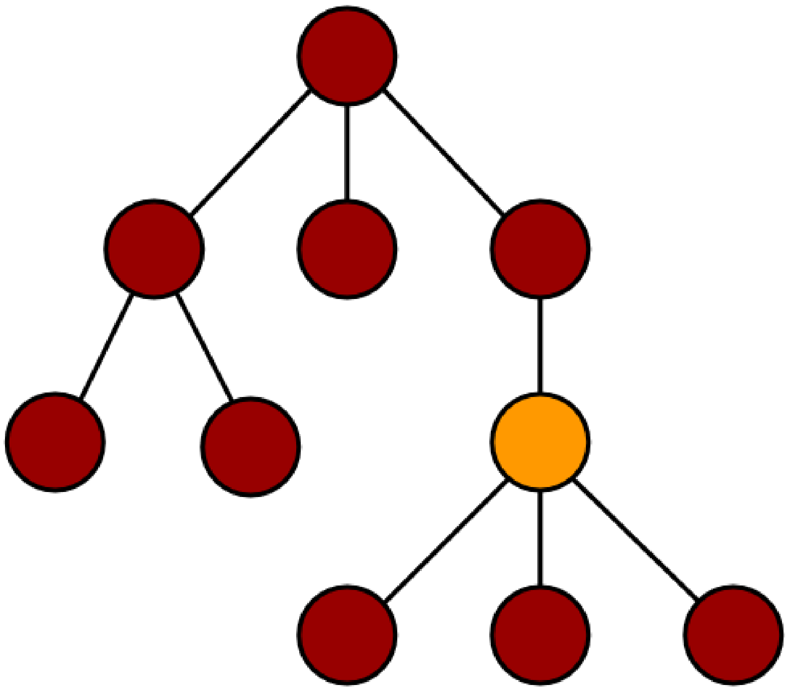
3.2 Rendering

In order use the virtual DOM, there has to be a way to convert virtual elements into DOM nodes. Therefore, a `render` function is introduced. This function recursively travels the virtual DOM and uses the DOM API to build up nodes.
Instead of directly manipulating the DOM with its API, a virtual DOM can be created including the needed changes. The only direct interaction with the DOM is when the newly rendered nodes are being attached to it.

One way for this to happen is as a response to an event.

```
const handleClick = $table => _ => {
  const items = ['Puerro', 'Huerto']; // fetching items from API's, DOM elements
or others
  const vDOM = createVDOM(items);
  $table.replaceChild(render(vDOM), $table.firstElementChild);
  return vDOM;
}
```

This, however, doesn't differ a lot of using `$table.innerHTML`. All the nodes are still getting re-created from scratch and previously held references are lost. It could also lead to a bad performance when dealing with huge DOM trees.
This illustration shows how rerendering works, when the orange node changed but every node is getting rerendered.



3.2.1 Identity Problem

Another problem when completely rerendering a large amount of elements is that they lose their identity. This is noticeable when a selected element or an element containing temporary state is being rerender. In that case, the focus is being lost which is not good from a usability point of view.

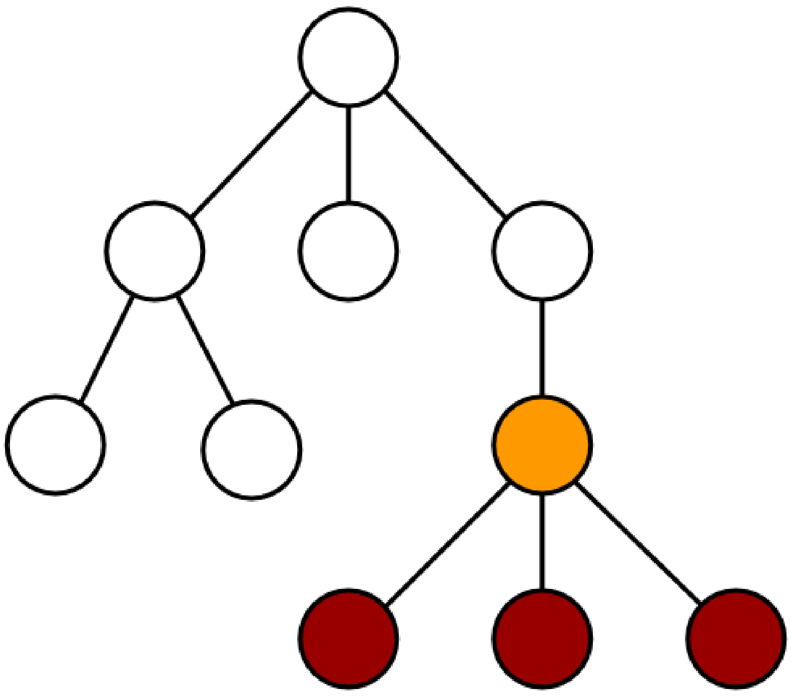
This could for example be the case inside a form, when the structure of the form changes based on the input and the whole form is being rerendered.

One possible solution for this problem is to save the identity before rerendering and manually applying it back afterwards. Another approach is to use diffing and only update elements which need to change.

3.3 Diffing

The real advantage of the virtual DOM can be seen when diffing is being used to only specifically update the parts and elements which have been changed.

With diffing the idea is to find the virtual nodes which have been changed and only rerender the parts of the tree which are necessary. This illustration shows how diffing works when the orange node has changed and only the colored nodes which need to be updated are getting rerendered.



In order to make this work, a diffing algorithm is needed to check the changes between two virtual DOM's and applying the changes to the actual DOM. Puerro has it's own diffing implementation.

```
diff($parrent, newVDOM, oldVDOM);
```

This implies that a *current* or *initial* version of the virtual DOM must exist. Since the state of this *current* version lives in the DOM, it needs to be converted into a virtual DOM first. Puerro provides the `toVDOM` function for this purpose.

```
const handleClick = $table => _ => {
  const vDOM = createVDOM(['Puerro', 'Huerto']);
  diff($table, vDOM, toVDOM($table.firstElementChild))
  return vDOM;
}
```

The result is that there are no more direct DOM interactions because the virtual DOM abstractions can handle them. Furthermore, only elements which really need to be changed are getting rerendered.

3.4 Testability

Using virtual elements results in a big benefit for testability. Instead of returning a DOM element and using the DOM API to test the content of the rendered view, the virtual DOM abstraction can be returned and tested with common JavaScript object approaches.

In case there is a specific need to test the DOM tree directly, the `render` function can be used to convert the virtual DOM into a normal DOM.

```
describe('vDOM', test => {
  test('createVDOM', assert => {
    // given
    const items = ['Puerro', 'Huerto'];

    // when
    const vDOM = createVDOM(items);

    // then
    assert.is(vDOM.children.length, 2);
    // possibility to interact via DOM API
    assert.is(render(vDOM).querySelector('td').textContent, 'Puerro');
  });
});
```

The described example can be found in the Puerro Examples.

3.5 Use Cases

The virtual DOM is useful when multiple elements need to be changed simultaneously or often. Instead of directly selecting and manipulating DOM nodes, the structure of the view can be written in a more descriptive way and the access to the DOM API is getting delegated to the general implementation of the virtual DOM.

Some possible use cases are:

- SPA (Single Page Applications) with huge DOM trees.
- When the DOM needs to change constantly and a lot.
- To display dynamically received content (e.g. over an API) without the need to store state.

3.5.1 Advantages

- No direct DOM manipulations.
- Declarative programming style.
- Reusability / less redundancy.
- Better testability.

3.6 Problems / Restrictions

The virtual DOM is an abstraction of the DOM. Operating on the virtual DOM is in addition to the DOM manipulations, which creates a computation overhead. This is especially the case for the diffing algorithm.

As in most abstractions, simplicity comes with the price of reduced flexibility. Meaning that it is not possible to handle every single edge case scenario with the virtual DOM.

Furthermore, the virtual DOM requires to completely build up a virtual view with all its sub elements, even though most of the content might never change and could be coded directly into the HTML view.

Handling state in the frontend is still not controlled in a structured way and updates on the view need to be triggered manually.

3.6.1 Disadvantages

- Computation overhead.
- Reduced flexibility.
- Less readable than HTML markup.
- Missing whitespaces between inline elements.
- Possible problems with the identity of elements.
- State lives in the view.

4 State Management

So far, the state of the application lived inside the view / the DOM and was not stored anywhere else. This might be acceptable for small applications which more or less only display data. As soon as a user is able to change and interact with data, a need arises for storing state in an organized manner. That is where state management comes into play.

4.1 State Inside the View

In previous chapters the state always lived inside the view and was not used further. In the following example there is a form with an `input` element, where the user can insert their name. The question which poses itself now, is how to handle and where to store this name?

```
const setName = evt => {
        // what to do with the new name?
}

const vDOM = h('form', { },
        h('input', { name: 'name', onInput: setName }),
);

const render = () => {
        // handle rendering; in this case with the virtual DOM
}
```

Until now the name would have been rendered right back out to the user but what if the goal is to store it and use this data later?

4.2 Keep it Simple

The simplest way to do state management in the frontend, is storing data in an object. Setting and retrieving state is very straight forward:

```
const myState = { name: '' };
const setName = evt => {
        myState.name = evt.target.value;
}
```

Although the state can now be updated, there is currently no way to get notified if it changes. The notification is imperative since the state might be used in different parts of the application.

While this approach is simple, it is also very limited and not useful in most situations, since updates have to be done manually in each event listener function.

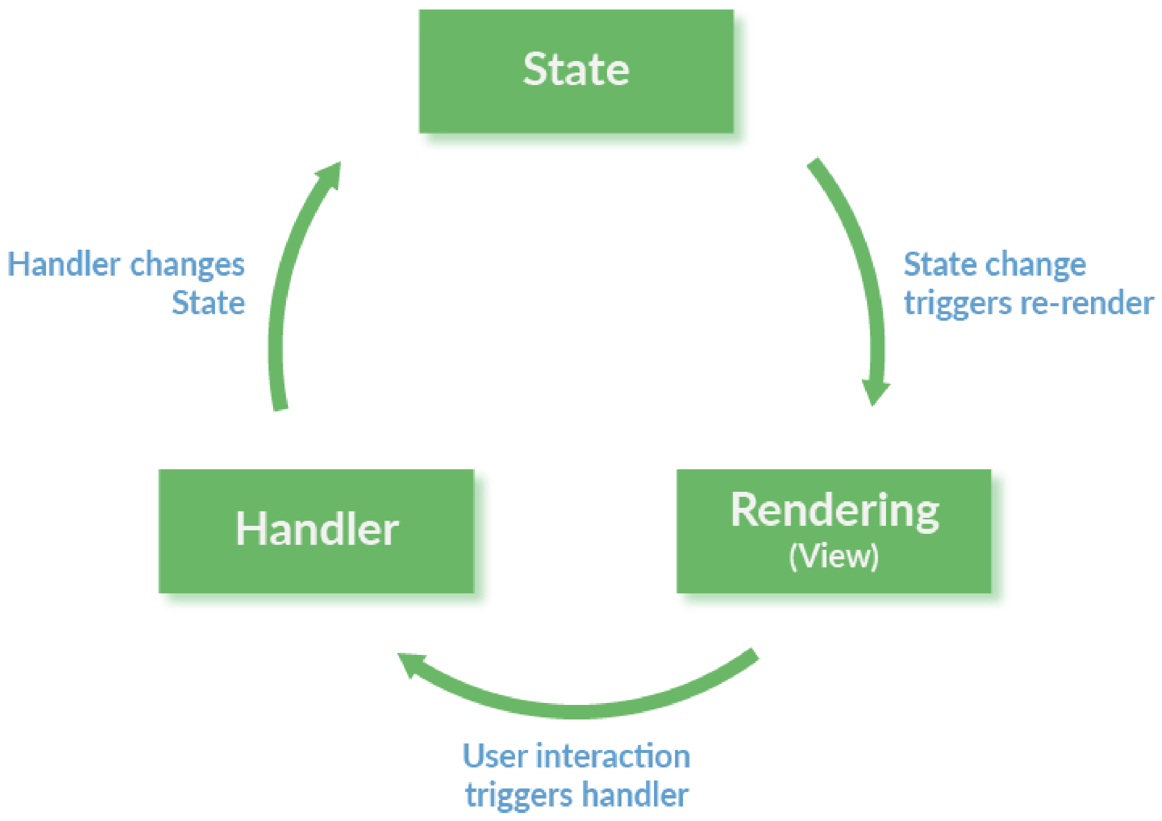
4.3 Components

Especially in combination with the virtual DOM, a powerful way to manage state, is to introduce components. Each component encapsulates a part of functionality which is used in the application. The components should not only provide a way to keep state but also be able to take actions in form of rerendering if the state changes.

Puerro provides a way to mount components and provide them with state. Check it out.

4.3.1 Render Cycle

With each action on the state the component needs to rerender, this leads to a rendering cycle which looks like the following:


State
State change triggers re-render
Rendering (View)
User interaction triggers handler
Handler
Handler changes State

With the state management in place as a dependency, a new component only has to define what the initial state is and how this state gets rendered. The following example uses the abstraction Puerro provides:

```
import { h, mount } from 'puerro';

const intialState = {
  num1: 0,
  num2: 0,
}

const component = ({ state, setState }) =>
  h('div', {},
    h('input', {
      type:  'number',
      name:  'num1',
      input: evt => setState({ num1: +evt.target.value })
    }),
    h('span', {}, '+'),
    h('input', {
      type:  'number',
      name:  'num2',
      input: evt => setState({ num2: +evt.target.value })
    }),
    h('span', { }, '= ' + (state.num1+state.num2)),
  );

mount(document.body, component, intialState);
```

Since the state is not mutated directly but set through the `setState` function, the component knows that it has to rerender. Inside the `setSate` function a few things happen:

1. The actual state object is set with the updated state.
2. The `component` function gets called with the new state.
3. The new virtual DOM from the `component` function is compared against the existing tree and updated in the DOM accordingly.

4.3.2 Web Components

A new concept in frontend development are web components. Web components is a generic term to describe a range of different new technologies to create reusable custom elements.

A custom element is created by first creating a class using the ES2015 class syntax extending a HTMLElement. The new custom element can then be registered to the document using the `CustomElementRegistry.define()` function.
These custom elements do not have a render cycle built in natively but are only used to encapsulate logic. Puerro provides a combination of state management and custom elements which results in highly reusable components. The previously used example of a simple calculator can be found in the Puerro Examples. With the help of

custom elements, the input component used is abstracted and can then be reused like a normal HTML element.

```
class MainComponent extends PuerroElement {
  static get Selector() { return 'puerro-main' };

  constructor() {
    super({ num1: 0, num2: 0 });
  }

  render() {
    return h('div', {},
      h(PuerroInputComponent.Selector, {
        label: 'num1',
        valueChanged: evt => this.setState({ num1: +evt.detail })
      }),
      h('span', {}, '+'),
      h(PuerroInputComponent.Selector, {
        label: 'num2',
        valueChanged: evt => this.setState({ num2: +evt.detail })
      }),
      h('span', {}, '= ' + (this.state.num1 + this.state.num2)),
    )
  }
}
```

As described before, for the custom elements to work they have to be defined with the `customElements` API:

```
window.customElements.define(PuerroInputComponent.Selector, PuerroInputComponent);
window.customElements.define(MainComponent.Selector, MainComponent);
```

They can then be used like normal HTML elements with the previously defined selector.

```
<body>
  <puerro-main></puerro-main>
</body>
```

Although not yet supported in all browsers, according to caniuse.com custom elements are already supported for 86% of internet users. For all other browsers, polyfills can be used to guarantee the support.

4.4 ID Management

If data is stored on the client side but changes have to be reflected in a persistence system (like a database) on the backend, id management becomes a difficult task. This is commonly the case for CRUD like applications. For the end user there is usually a master view in which all entries of an entity are shown and a detail view which displays and lets the user edit one selected entry. The problem which now arises is the creation of an ID. The question is, when must the ID be created?

Below are a few different approaches which can be considered.

4.4.1 Create an Empty Entry

One approach is to create an empty entry on the database and then let the user update this new entry. The problem with this strategy is, that the database initially contains an invalid entry which is hard to manage if for instance database constraints are in place.

4.4.2 Generate the ID on the Client-Side

Generating an ID can also be done on the client side and then sent along with the other properties of an entry to the backend. This approach works well if only one person uses the system but crumbles fast if there are multiple users. If two users create a new entry with the same database state, they will generate the same ID (if the ID is incremental and not probabilistic). This causes a conflict in the database which is hard to resolve.

4.4.3 Generate the ID on First Save

Another approach is to create the entry on the database as soon as it is first saved by the user. With this approach the ID is only known after the entry is sent to the backend once. It then has to be updated with this new ID for the master view.

4.5 Testability

Since the Puerro implementation of the state management uses the virtual DOM, the same advantages in testability apply to this chapter as well.

The described example can be found in the Puerro Examples.

4.6 Use Cases

State management becomes relevant relatively fast. As soon as data not only gets displayed but also has to be changed and stored, the need for some sort of state management arises.

Possible scenarios are:

- Applications which need to store data on the client side.
- Reactive applications which work with user interactions.

4.6.1 Advantages

This list refers to the Puerro implementation of state management.

- Automatic rerendering.
- Small amount of boilerplate code.
- Unidirectional dataflow.

4.7 Problems / Restrictions

With a managed state an application can become harder to manage and comprehend. It is important, that state is always stored in the same way to keep consistency and maintainability. This is a restriction that can be tedious for small projects.

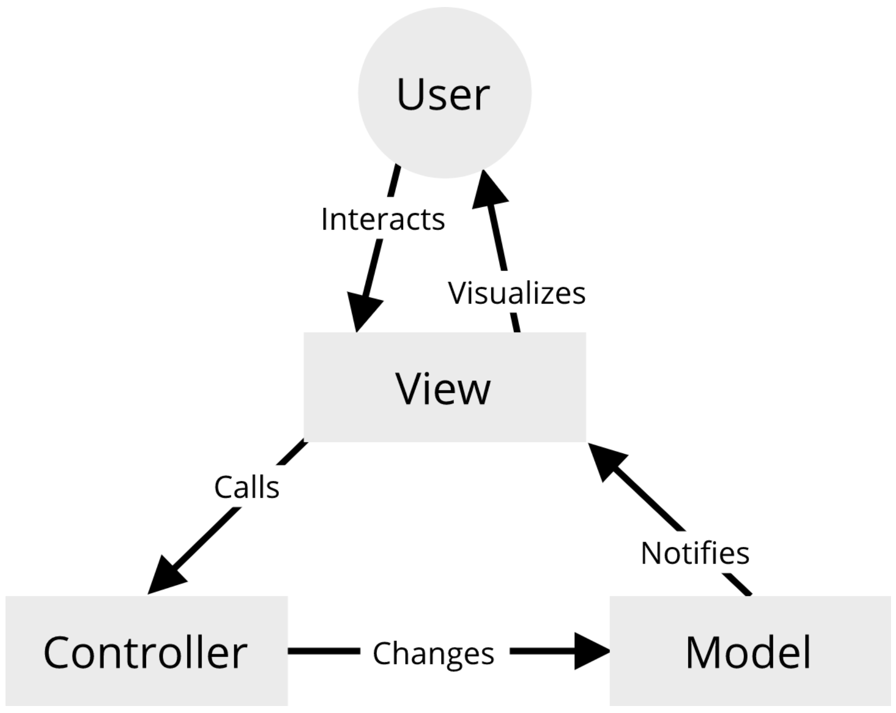
If the state updates the UI automatically, bugs can be hard to trace through the abstractions.

5 MVC

MVC has been around in the software development world for quite a while and it is a well-established way to create software. It is a type of application architecture which focuses very much on the separation of concerns.

5.1 The MVC Way

An MVC (Model-View-Controller) application is, as the name suggests, divided into *Model*, *View* and *Controller*. The following illustration represent how the data and events flow through the three components:



5.1.1 Models

A Model defines and stores data and is not concerned with how this data gets presented to the user. A model has to be observable, so when the data changes, the views get notified and can take action accordingly.

Puerro provides observables which are very useful for the definition of models. The following is an example of a model in an MVC approach with pure JavaScript (without frameworks, only the observable abstraction from Puerro):

```
import { Observable } from 'puerro';

const Model = ({ name = '', age = 0 } = {}) => ({
    name: Observable(name),
    age: Observable(age),
});
```

5.1.2 Views

The view is used to present the model to the user. It is the only component of a frontend MVC application which interacts with the DOM. The view subscribes to changes of the model and takes the required actions in the DOM to represent the new model state.

Users interact with the view which leads to events being fired from the DOM. Relevant DOM events are being listened to by the view and in turn passed to the controller if the user interaction requires a change to the model. The following code snippet is a view which could take the previous example model to represent data. The view also interacts with a controller which is introduced in the next chapter.

```
const View = (model, controller, $form, $output) => {
    const render = () =>
        ($output.innerText = `${model.name.get()} - ${model.age.get()}`);

    // View-Binding
    $form.name.addEventListener('input', evt =>
controller.setName(evt.target.value));

    $form.increase.addEventListener('click', controller.increaseAge);
    $form.decrease.addEventListener('click', controller.decreaseAge);

    // Model-Binding
    model.name.onChange(render);
    model.age. onChange(render);
}
```

Since the view does not concern itself with how the page is rendered, it can be done in different ways. In the previous example the DOM was manipulated directly, but this is just one way of handling the view layer. It can for example also be rendered with the virtual DOM which ensures all the benefits it brings. The following example shows how the virtual DOM can be integrated into a view:

```
const View = (model, controller, $form, $output) => {
    const view = () => h('div', {},
                           h('div', {}, model.name.get()),
                           h('div', {}, model.age.get())
                          );

    const render = name => {
        $output.replaceChild(renderVDOM(view()), $output.firstChild);
    }

    // same as before...
}
```

5.1.3 Controllers

The controller sits between models and views and is responsible for handling updates to the model. All actions which change the model have to go through the controller. Because all actions are handled by the controller, the application becomes very predictable and easier to reason about.

A controller for our example could look like the following:

```
const Controller = model => {
    const setName = name => {
        if (null == name || name.length === 0) {
            // handle invalid model state
        }
        model.name.set(name);
    }

    const setAge = age => {
        model.age.set(age);
    }

    const increaseAge = () => setAge(model.age.get() + 1);
    const decreaseAge = () => setAge(model.age.get() - 1);

    return {
        setName,
        increaseAge,
        decreaseAge,
    }
}
```

If the MVC architecture is combined with the revealing module pattern as shown in the above example, it is possible to use private functions like `setAge`. With this the interface of the controller is clearly defined and actions exposed to the view can be limited.
The controller is also the only place in the MVC architecture where side effects (for instance API calls) should be implemented.

5.1.4 Bringing it all Together

The Model, View and Controller can be separated into their own files and initialized in a central JavaScript file which is requested by the HTML.

```
const model = Model();
const controller = Controller(model);
const view = View(model,
                  controller,
                  document.getElementById('form'),
                  document.getElementById('output')
                 );
```

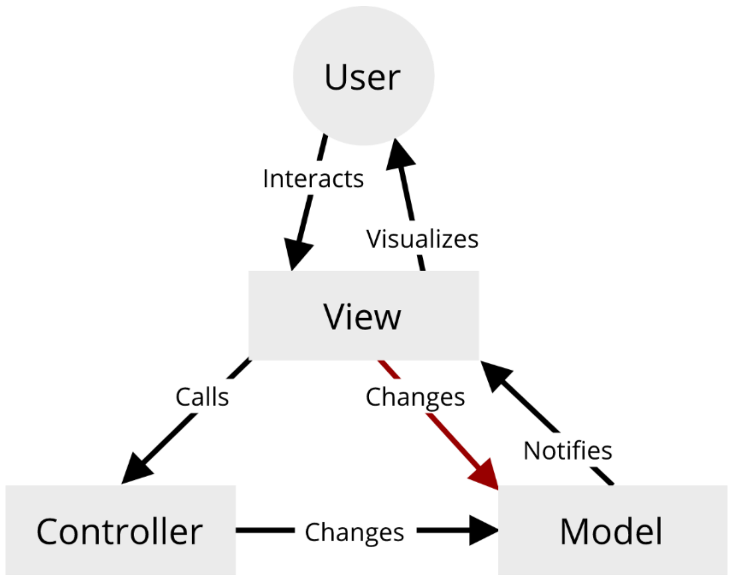
As evidenced by this example, MVC can generate some boilerplate code which seems over the top for small applications. But especially when the application scales, the advantages become very visible. The business logic is contained in one place and the model cannot be changed except through this business logic. Especially compared to the direct manipulation of the DOM and application state from anywhere in the code. This approach makes the application way more predictable and easier to understand.

5.2 Bidirectional Binding

Data and user inputs always flow unidirectionally in MVC. This makes the application easier to reason about and more predictable, since the model cannot be changed from anywhere.

Many modern frontend frameworks like Vue.JS and Angular (not React) work with bidirectional bindings of data. This is intuitive at first but as an application grows it can become very unpredictable. The model can be changed from anywhere and business logic has to be enforced with different approaches, for example in Vue.JS using watchers.

MVC does not permit bidirectional binding by design which might feel like a restriction sometimes, but it is essential to prevent bugs and keep the codebase understandable and maintainable. If it would allow bidirectional binding the graph would look like this:



This would defeat the whole purpose of the controller, by leaving it out. With this architecture we can never be quite sure that all the necessary business logic is executed.

5.3 Global State

The models hold the business data (or state) of our application. What if there are multiple separate MVC constructs which share a common state, but also have a state of their own? In this case, MVC in the frontend gets a bit tricky but there are a few ways to share state between controllers and models.

5.3.1 Client-Side Persistence Systems

In the backend data is usually persisted with a session or a database. With that in mind, there is an argument to be made to store the shared data in a persistence system which works on the client, for instance the `localStorage`. The downside to that is, that those systems are not reactive. This means that the values they store are not observable. Observable values are essential to ensure that the views get updated if an entry is changed. That is why those systems are not an optimal solution to handle global state.

5.3.2 Global Data Store

An alternative is to create a reactive global data store, which emits events, when data changes. This global store has to be accessible from every view and controller to retrieve and set values.

The Puerro library offers the `ObservableObject` constructor function.
An `ObservableObject` has the same functionality as an `Observable` with the addition that each property is separately subscribable as well.

```
import { ObservableObject } from 'puerro';

const myGlobalData = ObservableObject({ todos: [], userLoggedIn: false });

// subscribe to todo changes
myGlobalData.subscribe('todos', (newVal, oldVal) => {
        console.log(newVal); // logs todos
});

// subscribe to all changes to the store
myGlobalData.onChange((newVal, oldVal) => {
  console.log(newVal.userLoggedIn); // logs userLoggedIn
});

// add a todo:
myGlobalData.set({ todos: [...myGlobalData.get().todos, 'My new todo'] });
```

The rules of MVC still apply to the global store. It can only be changed from the controllers and not form anywhere else.

The downside to this approach is, that it can get confusing for a developer to figure out if data is stored in the global store or in the local model. It might seem obvious during the initial development of an application but can be very hard to manage while maintaining a codebase with a global store.

5.3.3 Multiple Views

MVC does not restrict one model and controller pair to be bound to exactly one view. A different approach to provide shared data between views is by injecting a model-controller pair into multiple views.

One could for instance extend the example used in the previous chapters and create a different view for the input and output of the application:

```
const InputView = (model, controller, $form) => {
    // View-Binding
    $form.name.addEventListener('input', evt =>
controller.setName(evt.target.value));

    $form.increase.addEventListener('click', controller.increaseAge);
    $form.decrease.addEventListener('click', controller.decreaseAge);
}

const OutputView = (model, controller, $output) => {
    const render = () =>
        ($output.innerText = `${model.name.get()} - ${model.age.get()}`);

    // Model-Binding
    model.name.onChange(render);
    model.age. onChange(render);
}
```

With multiple views, the rendering of different parts of our application can be splitted, but if the application gets larger the controller can get incomprehensible.

5.4 Testability

Business logic (in form of the controller) can be unit tested in a very efficient manner, since it is separated from the rest of the application. In previous examples we often had to mock DOM elements to test logic which can be tedious during the initial development but especially during maintenance.

```
import { describe }   from 'puerro';
import { Model, Controller } from './example';

describe('AppController', test => {
    const model = Model({ name: 'Test', age: 99 });
    const controller = Controller();

    test('setName empty', assert => {
        // when
        controller.setName('');

        // then
        assert.is(/* assert the error was properly handled */);
    });
});
```

Since in this case the intention is to only test business logic, there is not even a need to import the view because it doesn't matter how the data gets rendered into the DOM.

5.5 Use Cases

MVC can be used in many different situations but it shines the most in applications with a high amount of business logic. Since the business logic only lives in the controllers, maintaining and extending this logic becomes way easier than with other approaches. This also increases the overall stability of a system.

Possible scenarios are:

- Medium to large data driven applications.
- Applications which need to have a high maintainability.
- Business logic heavy applications.

5.5.1 Advantages

- Separation of concerns.
- Business logic and view are independently testable.
- High maintainability.
- Minimal amount of redundant code.

5.6 Problems / Restrictions

MVC requires quite a bit of boilerplate code, as well as some sort of observable. This can be an overkill for small applications and interactions.

5.6.1 Disadvantages

- Dependencies.
- Boilerplate code.
- Handling global state is hard.

6 Conclusion

The thesis for this project was, that big frameworks are not always needed for frontend Web Applications and that one can get very far by using a minimal amount of abstractions.

The work done in the showcase project "Huerto" shows that different problems call for different approaches. The more complex an application gets, the more helpful it is to use abstractions.

For large scale applications it is also very useful to hold to a specific architecture. In the Huerto example a variant of MVC is used. If used it is very important to hold to the principles given by the pattern used, to ensure consistency. Consistency is imperative in large scale web apps since it makes the codebase clean, maintainable and prevents a lot of bugs.

To make JavaScript code more testable it is advisable to separate business logic from DOM manipulation. That way the business logic can be tested independently and deterministically without depending on a specific environment.

Out of scope for this project were asynchronous side effects, like API calls. With such side effects and interaction with other systems web applications become even more complex and it is much more important to maintain a clean codebase.

Frameworks are not inherently bad! A good deal of frameworks have a big community which provides extensions and other solution for a lot of problems that might occur. Some frameworks are also backed and developed by big companies which gives them stability. Frameworks can be quite restrictive but that is also an advantage, it gives structure to a codebase. This structure is also good for teams with rotating members. For instance, a new hire which is proficient in Angular knows how an Angular project looks like and understands the code faster than they would with an application without a framework.

All that to say, frameworks tend to be overused and are not needed for every project. The dependency management alone can become a nightmare very quickly. That's why our recommendation is:

Begin with a zero dependency approach and add dependencies conservatively when the need arises.