SECTION 05 - VUE BEHIND THE SCENES

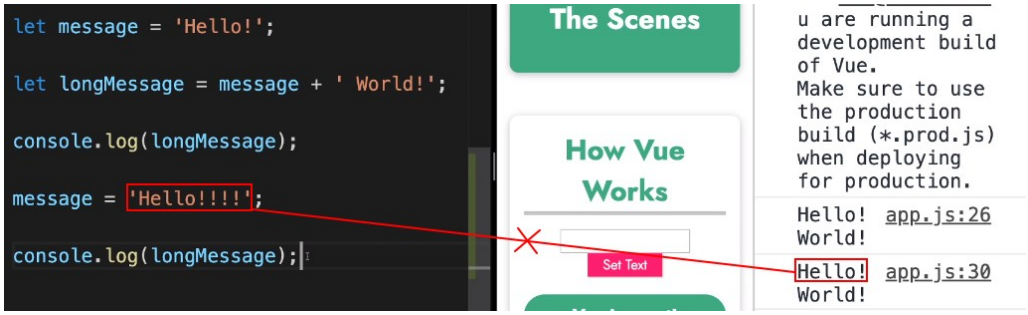
Index

- An introduction to Vue's reactivity

- Vue keeps track of the data you define in the object of the *return* from *data* method. All your properties and methods are merged into a global object.
- Vue also does other things like ensuring that *this* keyword points at that global object.
- Vue turns your data object into a ***reactive data object*** by essentially wrapping your properties with a JavaScript feature called ***Proxies***.
- With Proxy, a built in feature of JavaScript, Vue wraps your properties to be notified whenever you assign a new value.
- If Vue would not be notified about a new assignment, it could not go ahead and update what you see on the screen. It needs to be notified by changes so that it recognizes that it should scan the HTML code for a use. Then when it sees those usages and it knows that some variable changed, it goes out to the rendered page to the screen the user sees and finds the appropriate place where this dynamic content is displayed and updated there on the screen as well.

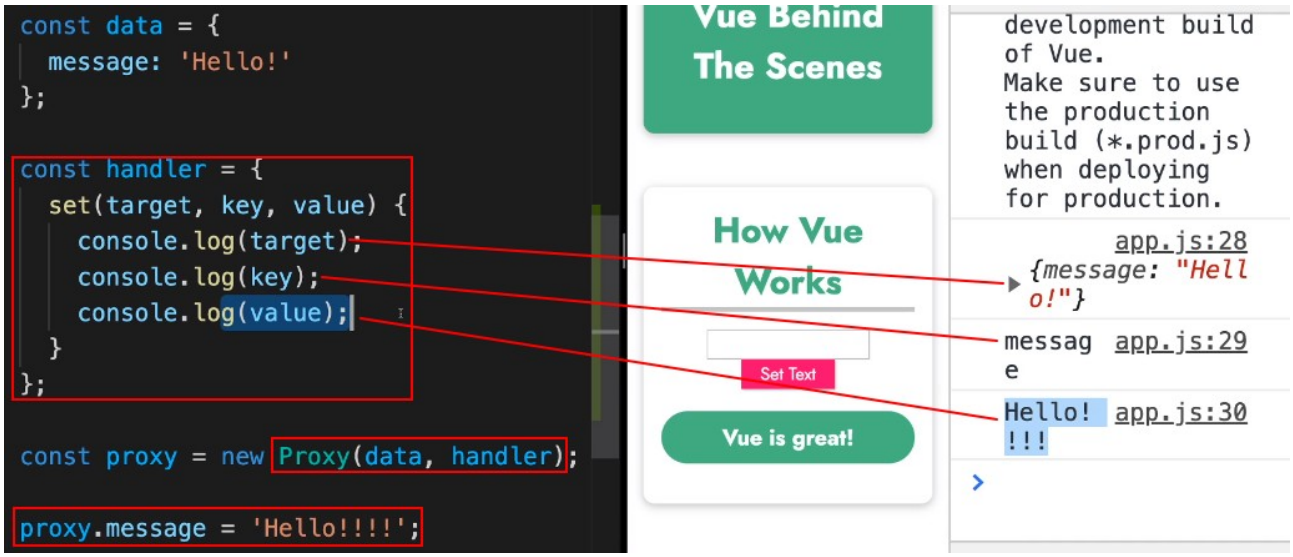
- Vue reactivity: A deep dive

- JavaScript by default is not reactive. If we have a calculation where we use a variable and we change that variable thereafter, that calculation is not re-executed. JavaScript doesn't note that variable changed and that it was used.



- The ***target*** is the object that was wrapped. The ***key*** is the property that you set a new value. And the ***value*** is the value that was set.

- We can define a *setter* function which is triggered whenever a property is set to a new value on this proxy.



- Reactive code using Proxy:

```
const data = {
  message: 'Hello!',
  longMessage: 'Hello! World!'
};

const handler = {
  set(target, key, value) {
    if (key === 'message') {
      target.longMessage = value + ' World!';
    }
    target.message = value;
  }
};

const proxy = new Proxy(data, handler);
proxy.message = 'Hello!!!!';
console.log(proxy.longMessage); // Hello!!!! World!
```

- We built our own reactive system for tracking one property (message) and updating another property (longMessage) when the first property (message) changed.

- And in a nutshell that is what Vue does under the hood. It keeps track of all your data properties and whenever such a property changes, it updates the part of your app where that property was used.

- One app vs multiple apps

- You can have one Vue app per page.
- Each view app works standalone, there is no connection between them.
- Multiple Vue apps are something you might want to use if you have different independent parts on your screen that should be controlled by Vue. If you have a part that needs to work together, though, for example, an input which is connected to a button, then this all should be controlled by the same app.

- Understanding templates

- Now, of course, it should be needless to say that you shouldn't control the same HTML part with different apps. And you also can't use one app to control multiple HTML parts. It's one HTML part per app.
- By mounting your Vue app to a certain place in the DOM, you make that part of the HTML code the template of that Vue app.
- There are different ways of defining your template. The most common (and most convenient one) is writing HTML code and then mounting your application to it. But it's not the only way of adding a template.
- You can also add a template to a Vue app by adding the template option to your app configuration object.

```
const app2 = Vue.createApp({
  template: `
    <p>{{ favoriteMeal }}</p>
  `,
  data() {
    return {
      favoriteMeal: 'Pizza'
    };
  }
});

app2.mount('#app2');
```

- Working with *Refs*

- I'm going to show you a different way of getting a value out of an input element.
- Thus far, we're listening to the *input* event, and therefore, with every keystroke, *saveInput* is fired, and in the *saveInput* method, the current value of the input is saved to other variable, so that we can eventually use that property later to assign the entered value to our message.

```
<section id="app">
  <h2>How Vue Works</h2>
  <input type="text" @input="saveInput">
  <button @click="setText">Set Text</button>
  <p>{{ message }}</p>
</section>
```

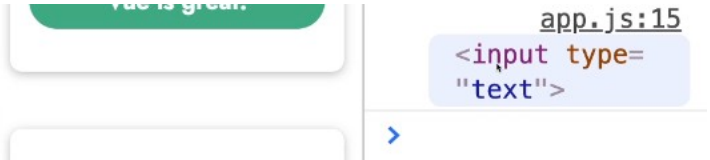
```
  data() {
    return {
      currentUserInput: '',
      message: 'Vue is great!',
    };
  },
  methods: {
    saveInput(event) {
      this.currentUserInput = event.target.value;
    },
    setText() {
      this.message = this.currentUserInput;
    },
  },
});

app.mount('#app');
```

- Whatever approach we use, we basically log what the user enters with every keystroke. Sometimes that is what you need, for example, because you want to validate the user input with every keystroke, but sometimes that's all just overkill, and in such cases, Vue has a feature that allows you to retrieve values from DOM elements when you need them, instead of all the time with ***refs***.

- Vue detects such refs and stores them internally. It basically memorizes that you want access to the input element, and in the JavaScript code you can then get this access.

- *this.$refs.userText* points at the DOM object for this input element. If we click on the button so that the *setText* function is triggered we'll see the DOM element printed:

```
setText() {
  // this.message = this.currentUser
  // this.message = this.$refs.userT
  console.log(this.$refs.userText);
```



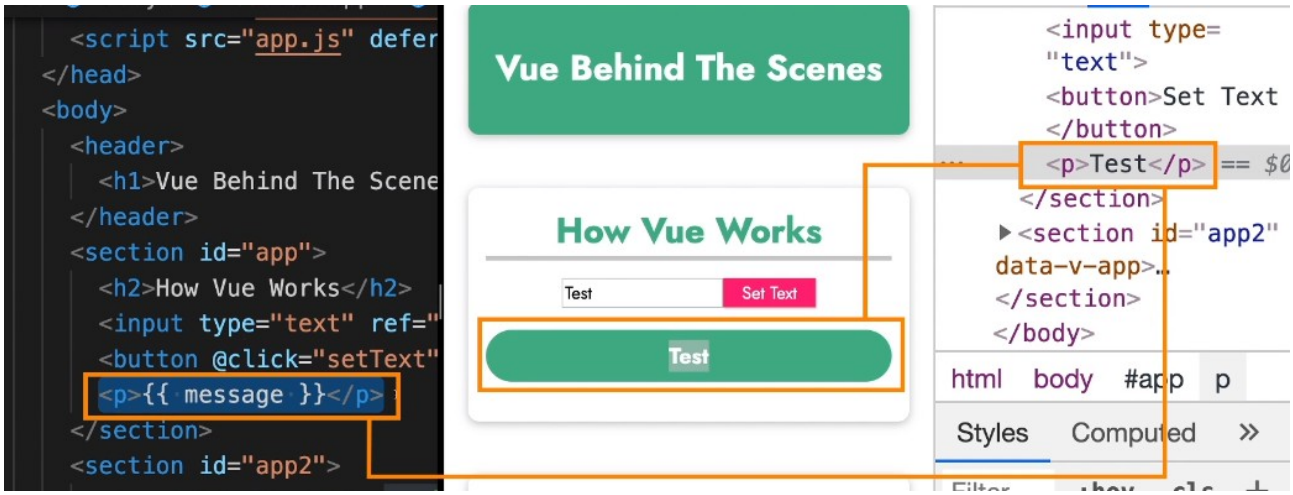

```
<section id="app">
  <h2>How Vue Works</h2>
  <input type="text" ref="userText">
  <button @click="setText">Set Text</button>
  <p>{{ message }}</p>
</section>
```

```
methods: {
  // saveInput(event) {
  //   this.currentUserInput = event.target.value;
  // },
  setText() {
    // this.message = this.currentUserInput;
    this.message = this.$refs.userText.value;
  },
},
```

- How Vue updates the DOM

- We got our Vue Instance, our view app, and we got the DOM, the Browser DOM. So the HTML content rendered by the browser. The Vue instance stores data computed properties, and methods. And the Browser DOM is then influenced with help of the view controlled template.

- That template content is rendered to the DOM. And during that rendering process which Vue controls, the dynamic parts and placeholders, the interpolations, and bindings, are removed and the actual values are inserted.



- Those Vue instructions are removed by Vue when it renders its template to the DOM. So when it basically converts its template, to the actual HTML elements that should be shown on the screen.

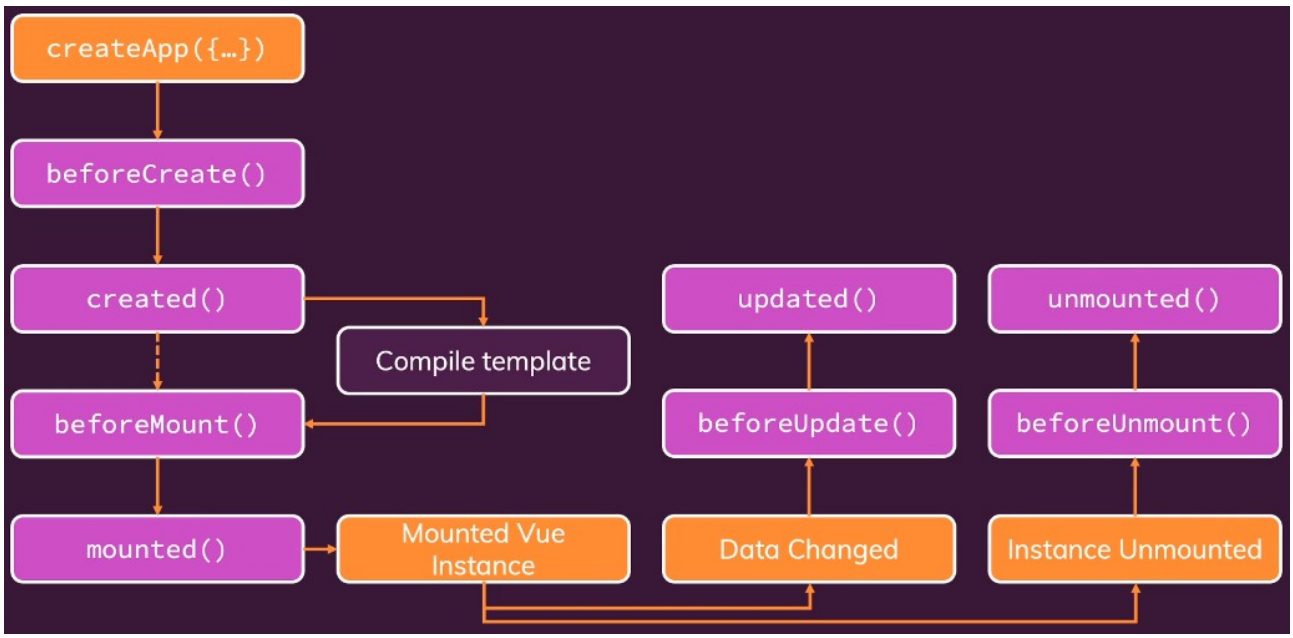
- Vue should not re-render the entire page every time something changes, but only the parts that did change. If Vue would reevaluate and re-render the entire page that would not be great for performance if just one piece of text changed.

- For re-rendering only the part that has changed, one possible solution would be that Vue compares the old DOM values with the new ones. So it basically goes through the entire rendered DOM and compares that content to the content it has stored in the data properties of the Vue app. If there's a difference, it updates it.
Vue could do that, and at least it would not update the entire screen, but actually it's not what it does because reading the entire real DOM whenever something changes is also pretty performance intensive.

- Vue uses a concept called the Virtual DOM.
- It has a virtual copy of the real DOM, which is managed entirely in JavaScript, and therefore in memory.
- When data changes, Vue creates a new virtual DOM and compares it to the old virtual DOM, and only differences between those virtual DOMs are written to the real DOM.
- Whenever we change something, for example if I change the text and click Set Text, Vue detects this change, thanks to the reactivity and then it creates a new Virtual DOM, compares it to the old Virtual DOM, and detects the differences. Then it updates the part in the real DOM where the differences were detected.
- Vue has a bunch of optimizations in place that actually make it way more performant. In reality, it does not recreate the entire virtual DOM's all the time. It has a bunch of optimizations for that.

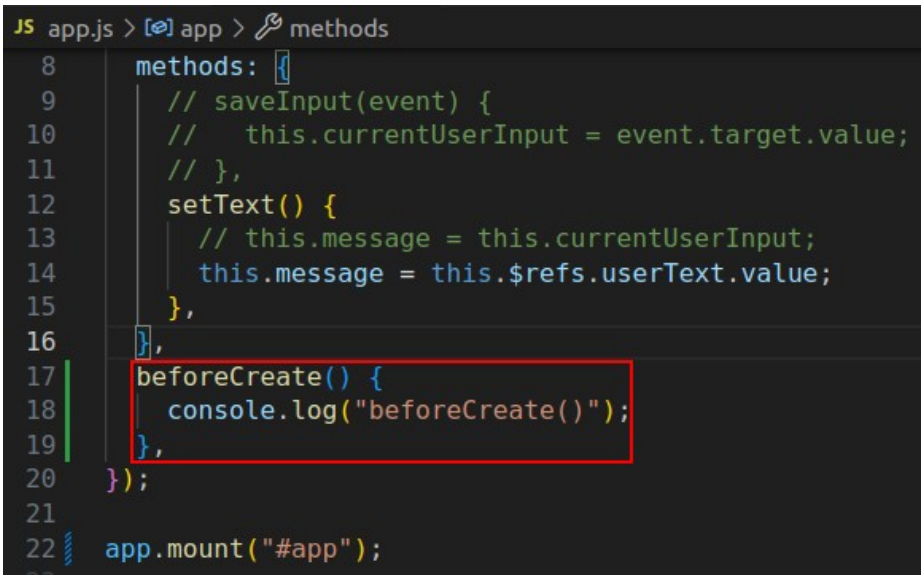
- Vue app lifecycle - theory

- The first lifecycle hook that is reached is the beforeCreate phase. It's followed by a created phase. The difference is that beforeCreate is basically called before the app has been fully initialized, whereas created is called thereafter. At this point of time, we still have nothing on the screen though. After created Vue just knows its data properties, and it's aware of the general app configuration.
- Then the template is compiled. All the dynamic placeholders, all the interpolations and so on are removed and replaced with the concrete values that should be shown to the user.
- Thereafter the beforeMount hook is reached. beforeMount means that this is right before Vue is going to bring something to the screen. So right before we can see something on the screen.
- Then it comes mounted. Now we see something on the screen. The Vue app was initialized, the template was compiled, Vue knows what to show on the screen. And it handed those instructions off to the browser, so that the browser adds all the HTML elements with all the content we need, as defined by our Vue app. Now we have our mounted Vue app, our mounted Vue instance.
- In most Vue apps data changes at some point of time, and that then triggers a new life cycle. Now we reach the beforeUpdate hook and there after the updated hook. This is a bit like beforeCreate and created. BeforeUpdate is reached before the update has been fully processed internally by Vue in its app, updated is reached, once that has been processed.
- Sometimes, an instance can also be unmounted. When an app is unmounted all its content is removed from the screen and the app is dead. And we have two lifecycle hooks here as well. BeforeUnmount, which runs right before it's about to be removed, and unmounted, which runs after it has been removed.



- Vue app lifecycle - practice

- For the beforeCreate method we could send an HTTP request to a server, set a timer, etc.



- If I put breakpoints in beforeMount and mounted methods, we first pause in beforeMount because that hook runs first. You see nothing on the screen. If you now click resume, we pause on the next break point, which is in mounted. Now you see something on the screen because at this point of time this Vue app was mounted to the screen. It was initialized internally, all the data was processed, the template was compiled and it's showing its output on the screen.

```
JS app.js > ...
 8 >   methods: {…
16     },
17     beforeCreate() {
18       console.log("beforeCreate()");
19     },
20     created() {
21       console.log("created()");
22     },
23     beforeMount() {
24       console.log("beforeMount");
25     },
26     mounted() {
27       console.log("mounted");
28     },
29   });
```

- If we add the other two hooks, in *beforeUpdate*, we'll not see the changes on the screen yet but in *updated*, we will.

- If we debug it adding some text in the input and clicking on the button, we first pause in beforeUpdate. And you notice that the text output here on the screen is still the old one. Because, in beforeUpdate the update was not fully processed yet. It's not visible on the screen. If I resume, we pause in updated. And indeed at this point of time, the output has been processed and is visible on the screen.