

Introduction to Vue 3 Training



with examples and
hands-on exercises

WEBUCATOR

Copyright © 2023 by Webucator. All rights reserved.

No part of this manual may be reproduced or used in any manner without written permission of the copyright owner.

Version: 1.1.0

The Authors

Chris Minnick

Chris Minnick, the co-founder of WatzThis?, has overseen the development of hundreds of web and mobile projects for customers from small businesses to some of the world's largest companies. A prolific writer, Chris has authored and co-authored books and articles on a wide range of Internet-related topics including HTML, CSS, mobile apps, e-commerce, e-business, Web design, XML, and application servers. His published books include Adventures in Coding, JavaScript For Kids For Dummies, Writing Computer Code, Coding with JavaScript For Dummies, Beginning HTML5 and CSS3 For Dummies, Webkit For Dummies, CIW E-Commerce Designer Certification Bible, and XHTML.

Nat Dunn (Editor)

Nat Dunn is the founder of Webucator (www.webucator.com), a company that has provided training for tens of thousands of students from thousands of organizations. Nat started the company in 2003 to combine his passion for technical training with his business expertise, and to help companies benefit from both. His previous experience was in sales, business and technical training, and management. Nat has an MBA from Harvard Business School and a BA in International Relations from Pomona College.

Follow Nat on Twitter at @natdunn and Webucator at @webucator.

Class Files

Download the class files used in this manual at

<https://static.webucator.com/media/public/materials/classfiles/VUE103-1.1.0.zip>.

Errata

Corrections to errors in the manual can be found at <https://www.webucator.com/books/errata/>.

Table of Contents

LESSON 1. Getting Started with Vue 3.....	1
Unpacking Vue 3.....	1
📄 Exercise 1: Vue 3 Hello, World!	2
Introducing Our Project: Mathificent.....	4
📄 Exercise 2: Get Started with vue-cli	7
📄 Exercise 3: Learning the Structure of a Vue App	11
LESSON 2. Basic Vue Features.....	19
The Vue Instance.....	19
Writing Vue Templates.....	21
📄 Exercise 4: Writing Templates	25
Using Components Inside Components.....	30
📄 Exercise 5: Breaking an App into Components	32
Passing Data to Child Components.....	41
Dynamic Data in Templates.....	44
Computed Properties.....	44
The data Object.....	46
The methods Object.....	46
Instance Lifecycle Hooks.....	50
LESSON 3. Directives.....	51
Directives.....	52
Conditionals with v-if / v-else-if / v-else.....	53
Two-way Binding with v-model.....	54
One-way Data Binding, Repeating, and Event Handling.....	58
Repeating an Element using v-for.....	59
Event Handling.....	62
Putting it All Together.....	63
Emitting Custom Events.....	65
LESSON 4. Implementing Game Logic.....	73
📄 Exercise 6: Passing Data Between Components	74
📄 Exercise 7: Vue Data Binding	80
📄 Exercise 8: Implementing Conditional Rendering	88
📄 Exercise 9: Improving the Form Layout	94
📄 Exercise 10: Making the Game UI	97
📄 Exercise 11: Capturing Form Events	109
📄 Exercise 12: Setting the Equation	115

LESSON 5. Transitions and Animations.....	127
Using the transition Component.....	127
📄 Exercise 13: Adding the Timer.....	131
📄 Exercise 14: Adding Transitions.....	138
📄 Exercise 15: Catching Keyboard Events.....	143
LESSON 6. Vue 3 Routing.....	145
Routing.....	145
Vue Router.....	147
📄 Exercise 16: Implementing Routes.....	150

LESSON 1

Getting Started with Vue 3

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ Starting a new Vue project.
- ☒ Structuring a Vue project.

Introduction

Vue 3, often just called “Vue,” is a progressive, incrementally adoptable, reactive front-end JavaScript framework. You’ll learn what that means in this lesson and you’ll also get started on your first Vue application.

EVALUATION COPY: Not to be used in class.



1.1. Unpacking Vue 3

One thing that makes Vue different from other front-end JavaScript frameworks is its philosophy of allowing incremental adoption. You don’t need to buy into an entire ecosystem with Vue when you first get started. It’s easy to just include the Vue library in a page and start using it. As you become more skilled with it, you’ll want to integrate it into a modern front-end development build, testing, and module structure, but, as you will see, Vue makes that process painless.



Exercise 1: Vue 3 Hello, World!

🕒 10 to 15 minutes

In this exercise, you will make your first Vue application by simply including Vue 3 in a page using a script element.

1. In the `ClassFiles/Vue/Exercises` folder, create a new file named `index.html` and write a basic HTML page containing a `div` element with an `id` attribute with a value of “app”:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>First Vue</title>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

Evaluation
Copy

2. Add the following script element in the head element of your new file to include the Vue library:

```
<script src="https://unpkg.com/vue@3"></script>
```

3. Add the following script element right above the closing `</body>` tag:

```
<script>
  const { createApp } = Vue
  createApp({
    data() {
      return {
        message: 'Hello, Vue!'
      }
    }
  }).mount('#app');
</script>
```

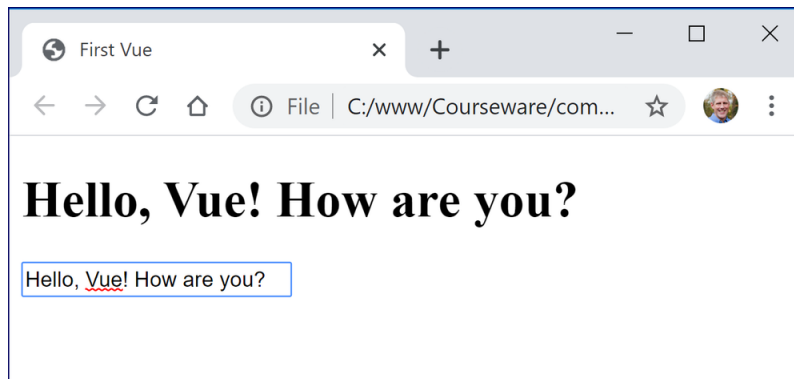
4. Type the following h1 element between the opening and closing `<div>` tags to dynamically display the `message` variable using Vue:

```
<div id="app">
  <h1>{{message}}</h1>
</div>
```

5. Save your HTML file and open it in a web browser. You'll see that `{{message}}` is dynamically replaced with the value that the `message` property of the `data` object was set to.
6. Type the following input element below the h1 in your HTML file, but still inside the `div` with the id of "app":

```
<input v-model="message">
```

7. Refresh (or reopen) `index.html` in your browser. Type into the input field and notice that the contents of the h1 are updated dynamically as you type:



Solution: Vue/Solutions/getting-started/index.html

```
1. <!DOCTYPE html>
2. <html lang="en">
3. <head>
4.   <meta charset="UTF-8">
5.   <meta name="viewport" content="width=device-width, initial-scale=1">
6.   <title>First Vue</title>
7.   <script src="https://unpkg.com/vue@3"></script>
8. </head>
9. <body>
10.  <div id="app">
11.    <h1>{{message}}</h1>
12.    <input v-model="message">
13.  </div>
14.  <script>
15.    const { createApp } = Vue
16.    createApp({
17.      data() {
18.        return {
19.          message: 'Hello, Vue!'
20.        }
21.      }
22.    }).mount('#app');
23.  </script>
24. </body>
25. </html>
```



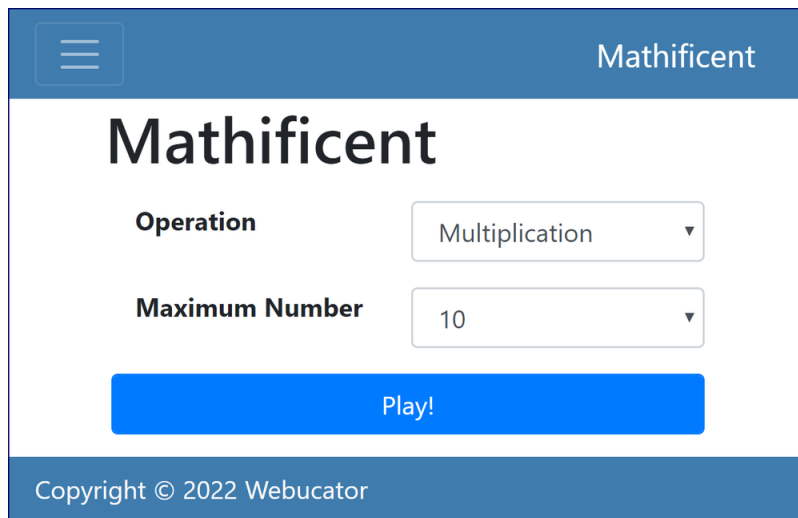
EVALUATION COPY: Not to be used in class.



1.2. Introducing Our Project: Mathificent

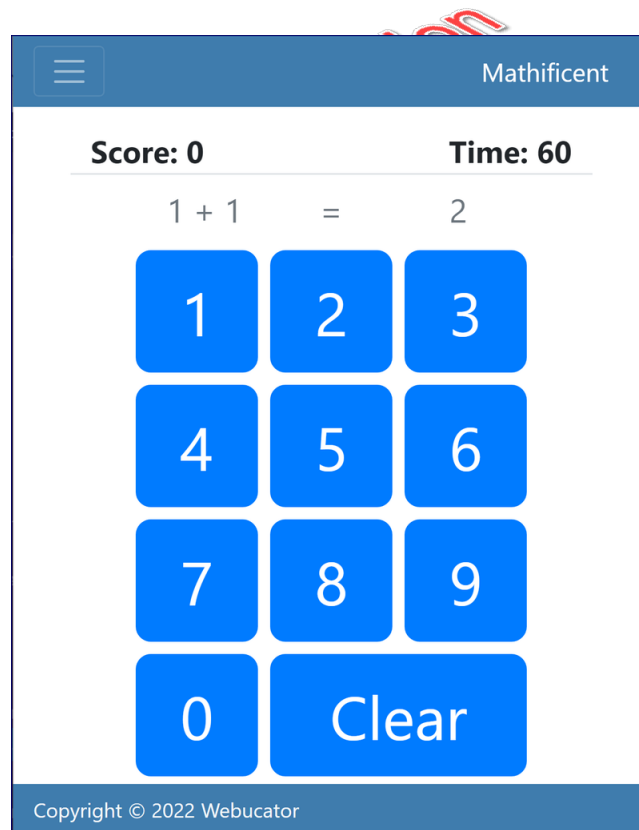
Throughout these lessons, you'll be using the latest Vue syntax and techniques to build a single-page application for practicing arithmetic. The application you'll be building is based on the game Mathificent, which you can view at <https://www.mathificent.com>. It consists of the following three views:

Config View



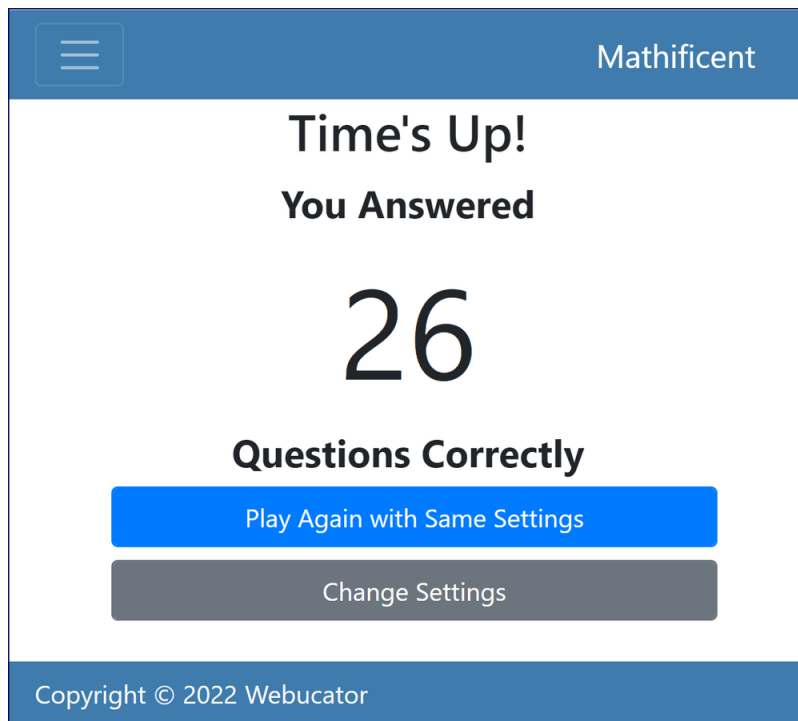
The Config View of the Mathifcent application features a blue header with a hamburger menu icon on the left and the text "Mathifcent" on the right. Below the header, the word "Mathifcent" is displayed in a large, bold, black font. Underneath, there are two configuration options: "Operation" with a dropdown menu set to "Multiplication", and "Maximum Number" with a dropdown menu set to "10". A prominent blue button labeled "Play!" is positioned below these options. At the bottom of the interface, a blue footer bar contains the text "Copyright © 2022 Webucator".

Game View




The Game View of the Mathifcent application features a blue header with a hamburger menu icon on the left and the text "Mathifcent" on the right. Below the header, the current "Score: 0" and "Time: 60" are displayed. A math problem, $1 + 1 = 2$, is shown. Below the problem is a numeric keypad with blue buttons for digits 1 through 9, 0, and a "Clear" button. At the bottom of the interface, a blue footer bar contains the text "Copyright © 2022 Webucator".

Times-Up View



Exercise 2: Get Started with vue-cli

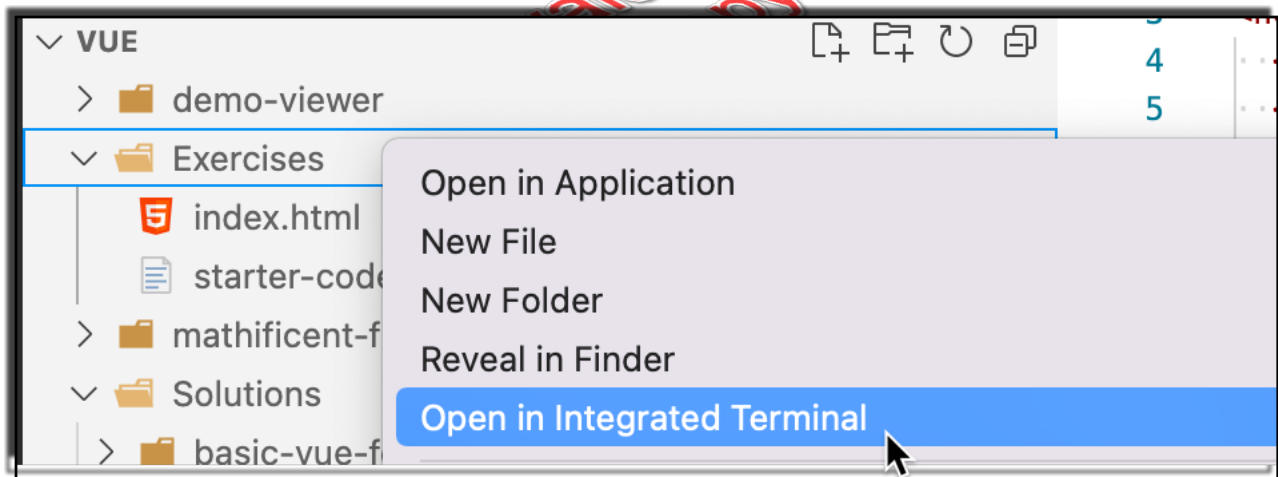
 10 to 15 minutes

Including Vue in your HTML files and writing JavaScript in `script` blocks, as you did in the previous exercise, works for very small applications and demos. However, Vue developers use a tool called *vue-cli*, which makes managing larger applications easier and sets up a development environment for compiling and working with Vue components.

In this exercise, you will use *vue-cli* to create your first Vue application, which will serve as the starting point for the math game you will be building with Vue. After you've built your Vue application, you'll use Node package manager (npm) to package and deploy the application to a development server.

Most of the process of building a simple application and installing the Node packages and scripts that make it run is done by *vue-cli*. This makes it easy to quickly start working on a project. So, let's jump in!

1. From your class files, open `Vue/Exercises` in the terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



2. Install *vue-cli* by running:

```
npm install -g @vue/cli
```

This will take a minute to install. If you get a message about vulnerabilities found, you can ignore it.

3. After vue-cli is installed, create the **mathificent** application by running the following command:

```
vue create mathificent
```

The script will ask you to make a choice regarding your development dependencies:

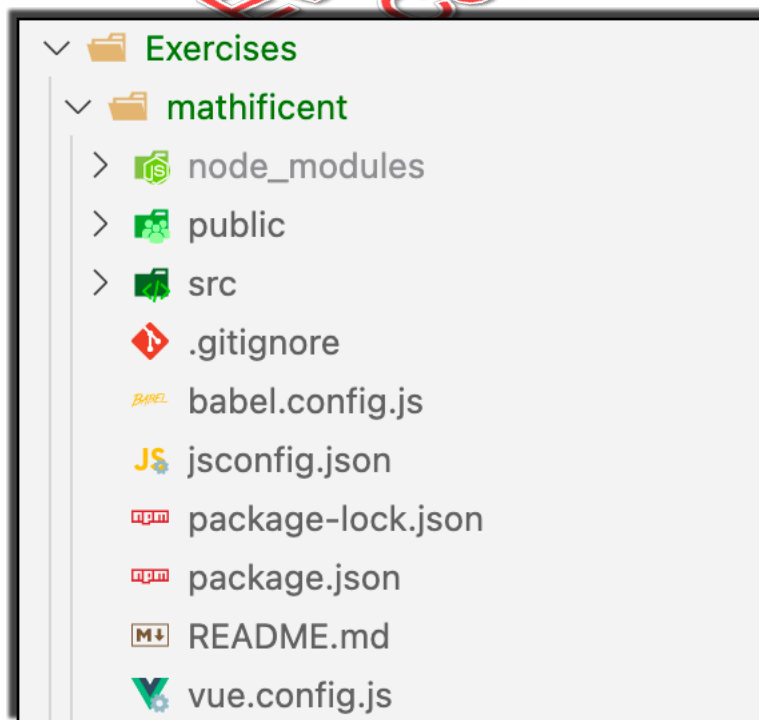
```
Vue CLI v5.0.6
? Please pick a preset: (Use arrow keys)
  Default ([Vue 3] babel, eslint)
  Default ([Vue 2] babel, eslint)
  Manually select features
```

We'll be using the default choice, so you can just press **Enter** when the question appears. If you are asked whether you want to use Yarn or npm to install dependencies, use the arrow keys on your keyboard to highlight **Use npm**, and then press **Enter**. The dependencies will be downloaded and after a few minutes you'll have a new Vue project.

4. Make your new Vue project the working directory:

```
cd mathificent
```

At this point, your first Vue program has been created and you can look at the individual files it contains:



5. Run:

```
npm run serve
```

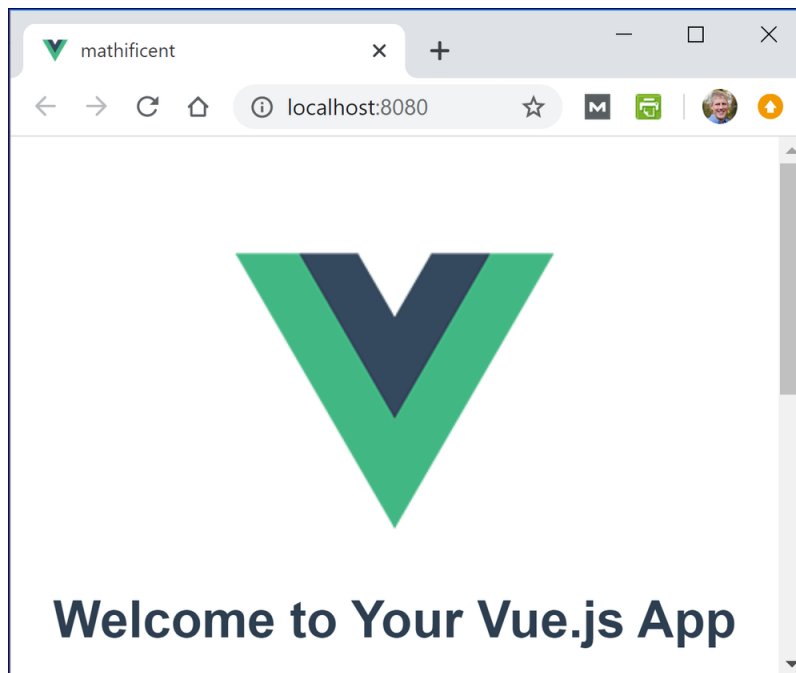
This is an npm script that was created when you ran `vue create`. Its job is to launch your Vue app using a development server. A development server is a web server that runs on a single software developer's computer and makes it possible for the developer to test out code as it is written and modified, without having to make it available for use by the entire internet.

6. When you see output similar to the following, it means your development server is running:

App running at:

- Local: `http://localhost:8080/`
- Network: `http://192.168.1.222:8080/`

Open a browser and go to `http://localhost:8080` to see your new Vue website:



7. Back in the terminal, press **CTRL+C** to stop the app. When prompted, confirm that you want to do so.
8. Close the terminal window by pressing the trash can icon:

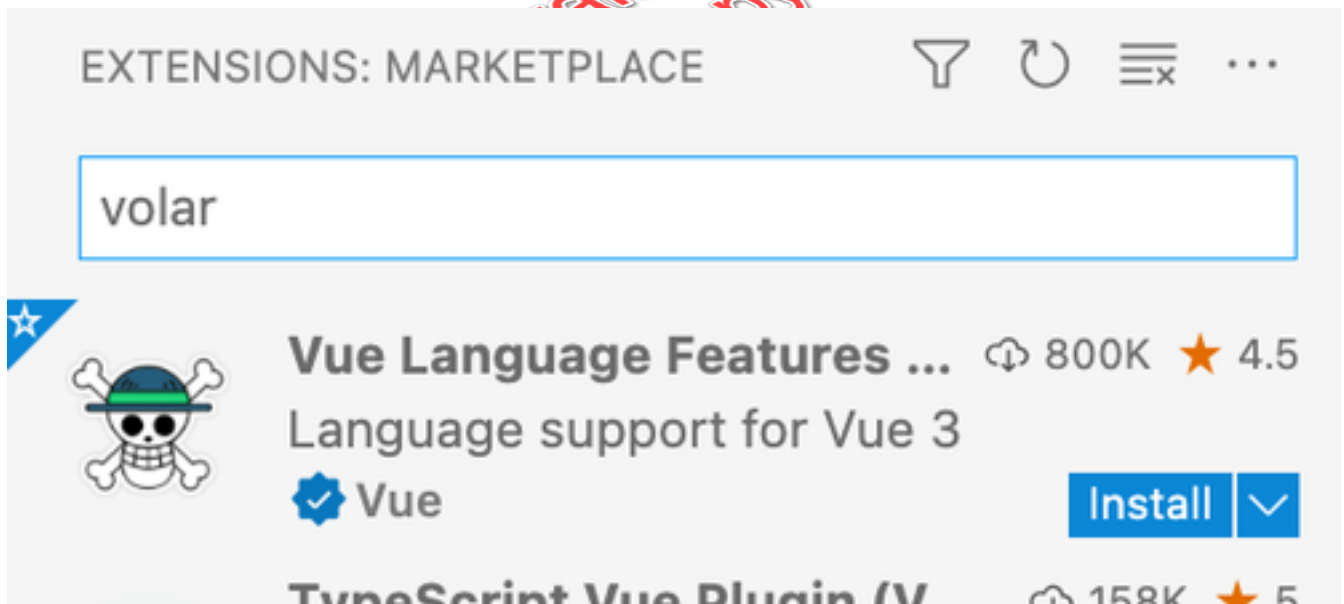


Exercise 3: Learning the Structure of a Vue App

🕒 20 to 30 minutes

In this exercise, you will start with a boilerplate Vue project and make some modifications to it to learn about the different files involved in Vue applications and the different parts of those files.

1. From the `mathificent` directory, open `src/App.vue` for editing.
2. By default, Visual Studio Code doesn't know how to properly highlight `.vue` files. To fix this problem, install the Volar extension in Visual Studio Code. It is possible that Visual Studio Code will prompt you to install this extension when you first open a file with a `.vue` extension. If it does, just click **Install** in the prompt. If you're not prompted, install Volar like this:
 - A. Click the **Extensions** icon (below the bug) on the left of the **Explorer** panel.
 - B. Search for "Volar".
 - C. Click the **Install** button.



3. When Volar is done installing, click the **Explorer** icon in the left panel of Visual Studio Code to return to viewing your project files. When you open a `.vue` file, it should now be properly color-coded.

4. Notice the structure of the `App.vue` file: it has a `template` block at the top, followed by a `script` block, followed by a `style` block. In Vue, this file is called a “single-file component.”
5. Examine the `template` block. It renders an image and another component, named `HelloWorld`:

```
<template>
  
  <HelloWorld msg="Welcome to Your Vue 3 App" />
</template>
```

6. Open `src/main.js` in your editor. This is the main JavaScript file for the entire Vue application. This file is the only place in your application that imports the Vue framework and renders the one component that contains every other component, also known as the *root* component. In our application, the root component is `App.vue`:

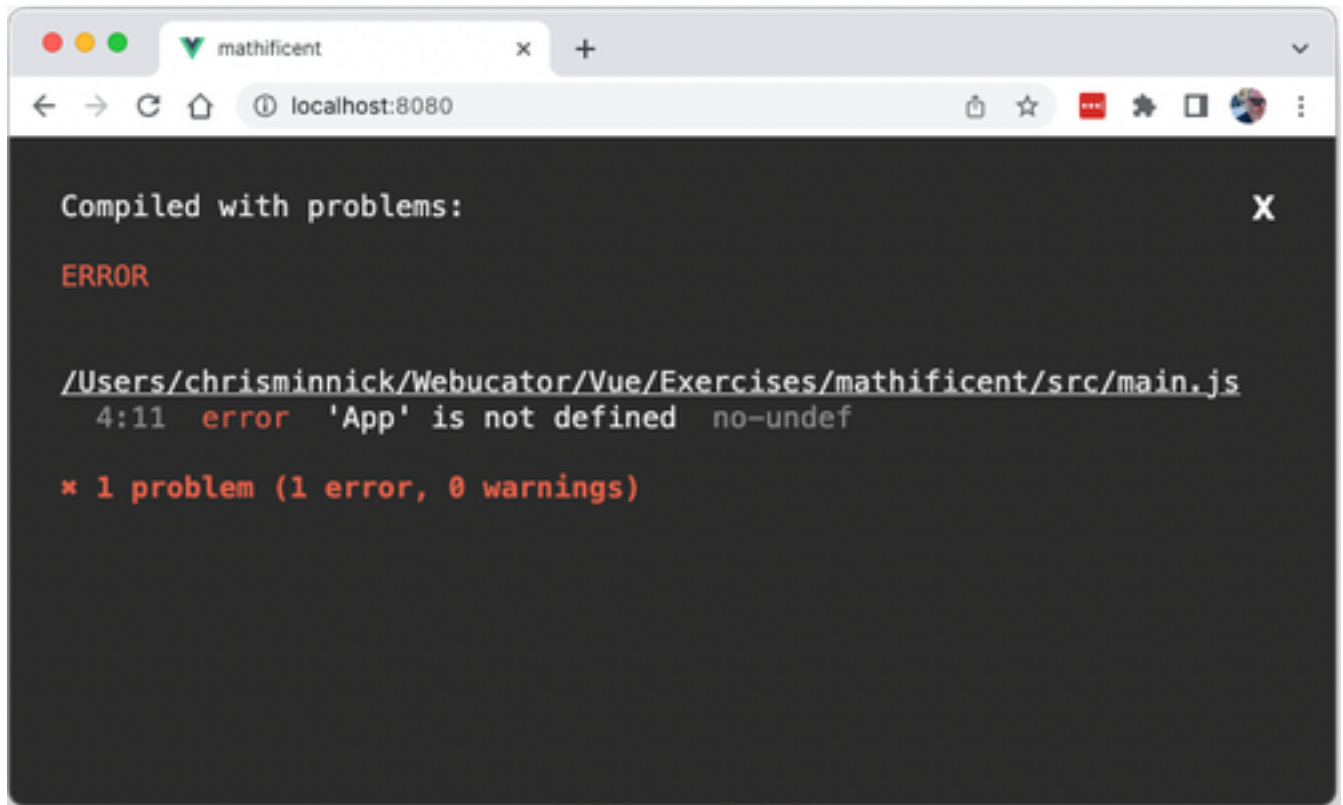
Exercise Code 3.1: `main.js`

```
1. import { createApp } from 'vue'
2. import App from './App.vue'
3.
4. createApp(App).mount('#app')
```

7. In `main.js`, comment out the `import` statement that imports `App`:

```
// import App from './App.vue'
```

8. Save your file.
9. If your development server isn't already running, start it by running `npm run serve` from the `mathifificent` directory.
10. Once the development server starts up, go to `http://localhost:8080` in your browser, where you'll see an error message similar to this one:



Because App is not imported, the reference to it on line 4 causes the application to fail to compile.

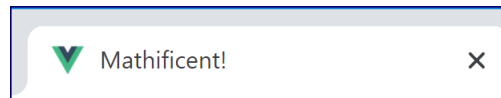
11. Return to your editor and remove the single-line comment before the `import` statement and save the file.
12. Return to your web browser and the application should refresh and be working again. Because Vue applications are made up of components that are linked together using `import` statements and any one file may have many `import` statements, one of the most common errors that you'll see in Vue development is caused by a component or file not being imported or not being imported correctly.
13. Look at the statement that begins with `createApp(App)`:

```
createApp(App).mount('#app')
```

The first part of this statement passes our root component into the `createApp` function to create a new application instance. However, an application instance won't do anything unless it's *mounted*. To mount the root component, the second part of this statement (after the period) calls the application instance's `mount` function. The argument passed to the `mount`

function is the location in the HTML document where the root component should be mounted. In this case, it's the element with an `id` attribute value of `app`.

14. Open `public/index.html` in your editor. This is the HTML file that is loaded when your web browser loads `http://localhost:8080`.
15. Change the title from `<%= htmlWebpackPlugin.options.title %>` to `"Mathificent!"` and save the file. Notice that the title updates on the browser tab:



16. Find the `div` element with the `"app"` id. This is where `index.js` will render the root component for your application:

```
<div id="app"></div>
```

17. Notice that `index.html` doesn't have any code that imports `main.js`. This is because `index.html` is a template. When you start the development server (using `npm run serve`), the code in `main.js` (and therefore everything that it imports) is injected into `index.html` with `<script>` tags before the page opens in your web browser.
18. Open `src/App.vue` in your editor.
19. Delete the `img` element and notice that the browser updates and the image of the Vue logo is gone.
20. Open `HelloWorld.vue` from the `src/components` directory. This file contains some links to useful Vue resources. It also contains a larger `template` block than the `App.vue` component. You don't need to do anything with this component, so close it when you're done examining the code, and then delete the `HelloWorld.vue` file from the `components` directory. This will cause the app to fail to compile. We'll fix that...

21. In `App.vue`, delete the `HelloWorld` component from the template block, delete the import statement that imports it, and delete `HelloWorld` from the `components` property of the export statement:

```
<template>
  <HelloWorld msg="Welcome to Your Vue 3 App" />
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>
```



22. In the template, add an `h1` element with the word “Mathifificent” inside it:

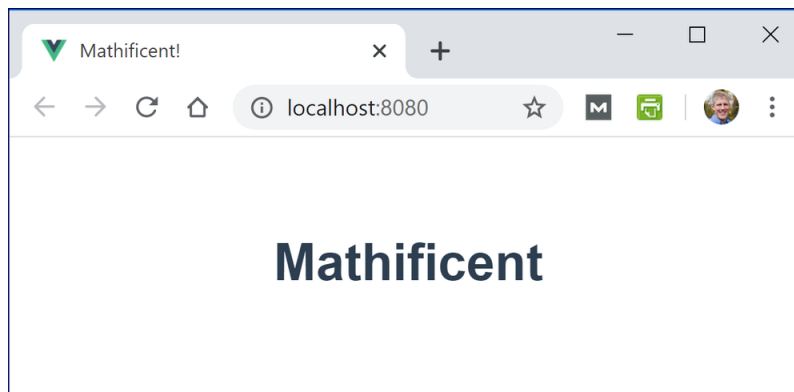
```
<template>
  <h1>Mathifificent</h1>
</template>
```

23. Your `App.vue` file should now look like this:

Exercise Code 3.2: App.vue

```
1. <template>
2.   <h1>Mathificent</h1>
3. </template>
4.
5. <script>
6. export default {
7.   name: 'App',
8.   components: {},
9. }
10. </script>
11.
12. <style>
13. #app {
14.   font-family: 'Avenir', Helvetica, Arial, sans-serif;
15.   -webkit-font-smoothing: antialiased;
16.   -moz-osx-font-smoothing: grayscale;
17.   text-align: center;
18.   color: #2c3e50;
19.   margin-top: 60px;
20. }
21. </style>
```

24. And your application should now look like this in the browser.



25. If you see an error message, return to your `App.vue` file and make sure that it matches the solution exactly.
26. Remember to stop the app (**CTRL+C**) and close the terminal when you are done.

Conclusion

In this lesson, you have learned how to make a basic Vue application by including the Vue library into an HTML page and by installing and running `vue-cli`.

LESSON 2

Basic Vue Features

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ Vue templates.
- ☒ Breaking a Vue app into components.
- ☒ Passing data between components.
- ☒ Dynamic data.

Introduction

In this lesson, you will build a form that allows the user to choose a math operator and a number. In doing so, you will learn how to work within Vue templates, how to break an application into components and pass data between those components, how to modify that data on the fly, and how to work with computed properties and methods.

EVALUATION COPY: Not to be used in class.



2.1. The Vue Instance

A Vue application consists of a root Vue instance created by invoking `createApp` and passing an *options* object into the `createApp` constructor function. In its most basic form, every Vue application looks like this:

```
const app = createApp({  
  // options  
})
```

Most often, the root Vue instance is broken up into components, which are each also Vue instances. Each of these components has its own *options* object as well.

The contents of the *options* object determines how the Vue application will work.

Options vs. Composition API

Vue 3 has two different styles for creating components. The one we'll be using in this course is called the Options style. The other style is called Composition. Both styles use the same underlying Vue system. However, the Options style is currently more widely used and is somewhat easier to program with. The Composition style is more free-form, and is better suited for large applications. To learn more about the differences between the Options style and the Composition style, visit the Vue documentation at <https://vuejs.org/guide/introduction.html#api-styles>.

❖ 2.1.1. Instance Properties and Methods

In addition to the custom properties and functions that you write, every Vue object has certain built-in instance properties and methods that give you access to information and functionality. These all begin with a “\$” sign to differentiate them from user-defined properties. We'll look at and use some of the instance properties and methods in the upcoming exercises. For now, the only thing you need to know is that any time you see a function or variable that starts with a “\$” in Vue, that is something that is baked into Vue.

❖ 2.1.2. Reactivity and Data

Each Vue instance contains a function called `data`. This `data` function is what makes Vue reactive. When the Vue instance is created, the `data` function runs and returns an object. Everything inside the `data` object is loaded into Vue's *reactivity system*. When one of the properties in the reactivity system changes, Vue updates the view (by merging the data into templates) to reflect the new value.

EVALUATION COPY: Not to be used in class.



2.2. Writing Vue Templates

Templates determine how a Vue instance will render. Templates are usually written using HTML, but they can also be written using JavaScript or an XML template language called JSX. Each Vue component has its own template, and these templates can contain other components as well as HTML and JavaScript.

You can use dynamic data and code in a template by surrounding it with double curly braces:

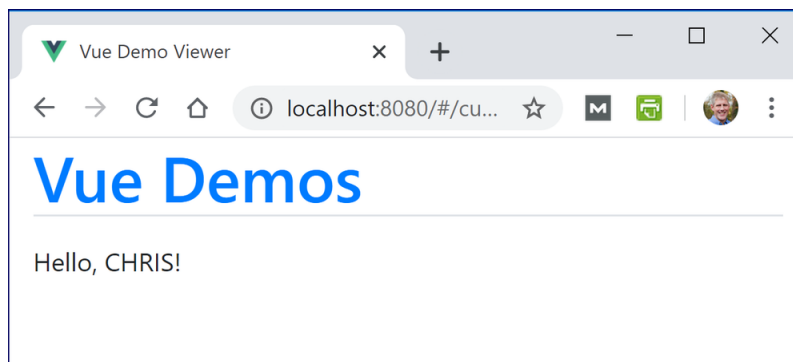
Demo 2.1:

Vue/demo-viewer/src/components/basic-vue-features/CurlyBraces.vue

```
1. <template>
2.   <p>Hello, {{firstName.toUpperCase()}}!</p>
3. </template>
4.
5. <script>
6.   export default {
7.     name: "CurlyBraces",
8.     data: function() {
9.       return {
10.        firstName: 'Chris'
11.      }
12.    }
13.  }
14. </script>
```

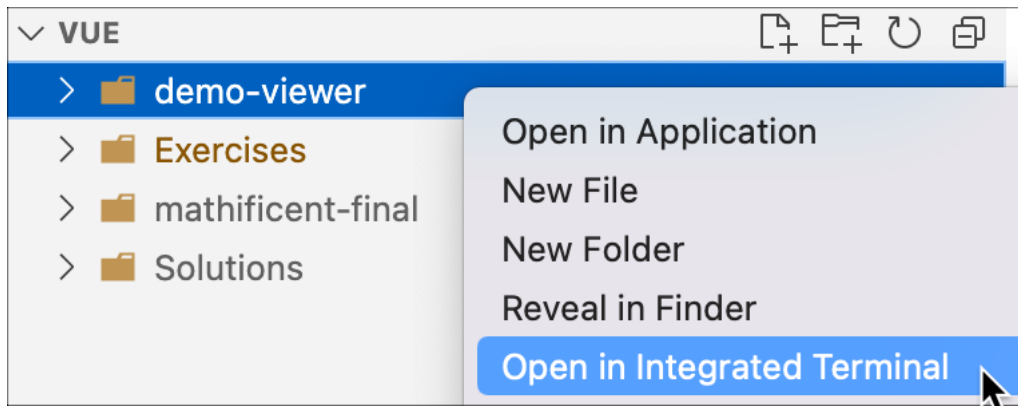
Evaluation
Copy

This creates the following page:



Notice that the content in the double curly braces is interpreted: instead of outputting “{{firstName.toUpperCase()}}”, it outputs the value of `firstName` in all uppercase letters. Don’t worry yet about the structure of the code in the `script` element. It is enough to see that `firstName` contains “Chris”.

1. To run this file, open `Vue/demo-viewer` in the terminal by right-clicking the folder and selecting **Open in Integrated Terminal**:



2. Install the demo-viewer app using `npm install`:

```
npm install
```

This will take a minute.

3. Once it has installed, run `npm run serve` and then open `http://localhost:8080` in your browser. Click the **Curly Braces** link under **Basic Vue Features**.

❖ 2.2.1. Curly Braces and v-html

Anything in between double curly braces will be output as plain text by default. For example, if a JavaScript variable contains less than and greater than signs, those signs will show up in their raw format instead of working as the beginnings and endings of HTML tags.

The `v-html` attribute (actually, it is a *directive* but we'll get into that later) is used to specify that some dynamic data should be output as HTML.

In the following example, the template will output a dynamically generated title, first using curly braces and then using `v-html`:

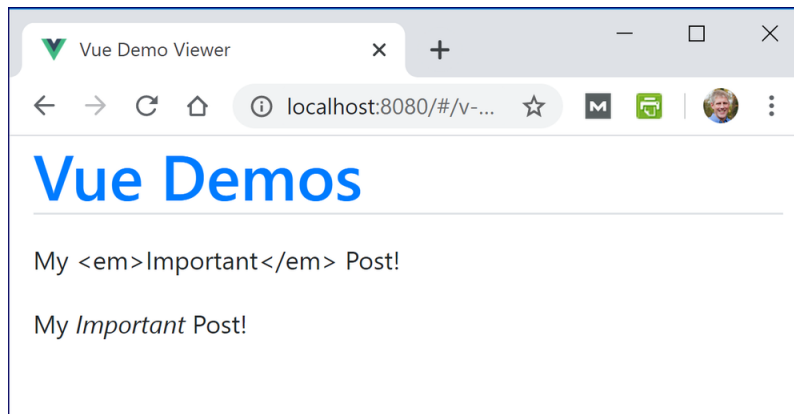
Demo 2.2:

Vue/demo-viewer/src/components/basic-vue-features/VHtml.vue

```
1. <template>
2.   <p>{{title}}</p>
3.   <p v-html="title"></p>
4. </template>
5.
6. <script>
7.   export default {
8.     name: "VHtml",
9.     data: function() {
10.      return {
11.        title: "My <em>Important</em> Post!"
12.      }
13.    }
14.  }
15. </script>
```

Again, don't worry about the script element yet, except to see that that is where title is defined.

If the demo-viewer app is still running, click the [Vue Demos](#) heading to get back to the home page. If it isn't already running, run `npm run serve` from the demo-viewer directory and then open `http://localhost:8080` in your browser. Then click the [v-html Directive](#) link under **Basic Vue Features**. You should see a page that looks like this:



Notice that the `` tags are output literally when the double curly braces are used, but they are kept as HTML tags when the `v-html` directive is used. The code Vue creates looks like this:

```
<p>My &lt;em>Important&lt;/em> Post!</p>  
<p>My <em>Important</em> Post!</p>
```

Remember to stop the app (**CTRL+C**) and close the terminal when you are done.



Exercise 4: Writing Templates

⌚ 20 to 30 minutes

In this exercise, you will continue building the Mathificent app. You will use HTML in Vue templates to create a user interface.

1. To add style to the app, we will use Bootstrap:
 - A. In Visual Studio Code, open `index.html` from the `Exercises/mathificent/public` directory.
 - B. Copy the following `<link>` tag:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-0evHe/X+R7YkIZDRvuzKMRqM+OrBnVFBL6DOitfPri4tjfHxaWutUpFmBp4vmVor" crossorigin="anonymous">
```

Paste it in the head in `index.html`.

- C. Copy the following `<script>` tag:

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/js/bootstrap.bundle.min.js" integrity="sha384-pprn3073KE6tl6bjs2QrFaJGz5/SUsLqktiwsUTF55Jfv3qYSDhgCecCxMW52nD2" crossorigin="anonymous"></script>
```

Paste it immediately before the close `</body>` in `index.html`.

2. Open `src/App.vue` in your editor.

3. Inside the `template` block, create a `<header>` element with the header navigation for our app by placing the following code above the `h1` element:

```
<header>
  <nav class="navbar navbar-expand-lg navbar-dark">
    <div class="container-fluid">
      <button class="navbar-toggler" type="button"
        data-bs-toggle="collapse" data-bs-target="#navbarText">
        <span class="navbar-toggler-icon"></span>
      </button>
      <div class="collapse navbar-collapse" id="navbarText">
        <ul class="navbar-nav mr-auto text-left">
          <li class="nav-item active">
            <a class="nav-link" href="/">Home</a>
          </li>
        </ul>
      </div>
      <a class="navbar-brand" href="/">Mathificent</a>
    </div>
  </nav>
</header>
```

A Shortcut: Copy and Paste

You can copy and paste from `Exercises/starter-code.txt` if you would prefer not to type this out. You will find both this header element and the footer element (shown below) in that document. If you do so, be sure to review both carefully, so you understand what's going on. They both include some Bootstrap classes, and the footer includes some JavaScript enclosed in double curly braces.

4. Under the `h1` element in the `App.vue` template, type the footer:

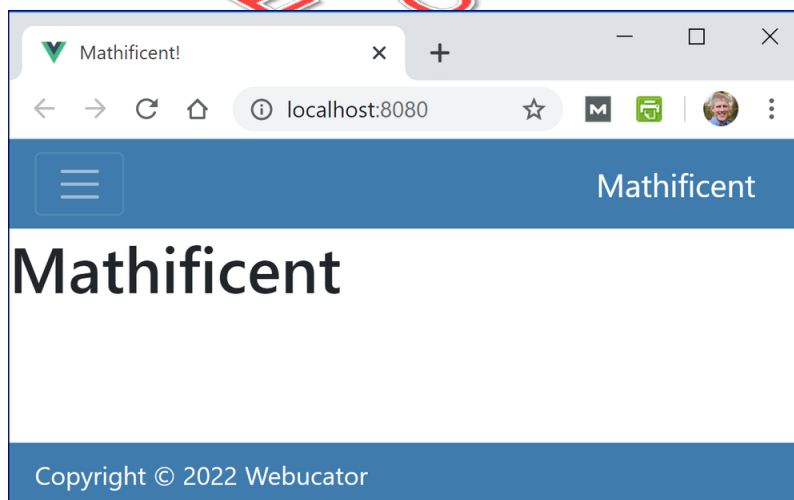
```
<footer class="navbar fixed-bottom">
  <div class="container-fluid">
    <a href="https://www.webucator.com" class="text-light nav-link">
      Copyright &copy; {{new Date().getFullYear()}} Webucator
    </a>
  </div>
</footer>
```

The JavaScript in the footer will dynamically populate the copyright year. Notice that it is enclosed in double curly braces. This lets Vue know that it should be interpreted.

5. In the style block at the bottom of `App.vue`, delete the existing rules and then add a rule to give the header and footer a steel blue background color:

```
<style>
footer,
header {
  background-color: #3f7cad;
}
</style>
```

6. Save `App.vue`.
7. Open the `mathificent` directory at the terminal and run `npm run serve` and then open `http://localhost:8080` in your browser. You should now have a header, an `h1` element, and a footer:



Solution: Vue/Solutions/basic-vue-features/writing-templates/App.vue

```
1.   <template>
2.     <header>
3.       <nav class="navbar navbar-expand-lg navbar-dark">
4.         <div class="container-fluid">
5.           <button class="navbar-toggler" type="button"
6.             data-bs-toggle="collapse" data-bs-target="#navbarText">
7.             <span class="navbar-toggler-icon"></span>
8.           </button>
9.           <div class="collapse navbar-collapse" id="navbarText">
10.            <ul class="navbar-nav mr-auto text-left">
11.              <li class="nav-item active">
12.                <a class="nav-link" href="/">Home</a>
13.              </li>
14.            </ul>
15.          </div>
16.          <a class="navbar-brand" href="/">Mathificent</a>
17.        </div>
18.      </nav>
19.    </header>
20.    <h1>Mathificent</h1>
21.    <footer class="navbar fixed-bottom">
22.      <div class="container-fluid">
23.        <a href="https://www.webucator.com" class="text-light nav-link">
24.          Copyright &copy; {{new Date().getFullYear()}} Webucator
25.        </a>
26.      </div>
27.    </footer>
28.  </template>
29.
30.  <script>
31.    export default {
32.      name: 'App',
33.      components: {
34.      }
35.    }
36.  </script>
37.
38.  <style>
39.    footer,
40.    header {
41.      background-color: #3f7cad;
42.    }
43.  </style>
```

The preceding code shows the complete `App.vue` file.

Solution: [Vue/Solutions/basic-vue-features/writing-templates/index.html](#)

```
1. <!DOCTYPE html>
2. <html lang="en">
3.   <head>
4.     <meta charset="utf-8">
5.     <meta http-equiv="X-UA-Compatible" content="IE=edge">
6.     <meta name="viewport" content="width=device-width,initial-scale=1.0">
7.     <link rel="icon" href="<%= BASE_URL %>favicon.ico">
8.     <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/css/boot  44
        strap.min.css" rel="stylesheet" integrity="sha384-0evHe/X+R7YkIZDRvuzKM  44
        RqM+OrBnVFB�6D0itfPri4tjfHxaWutUpFmBp4vmVor" crossorigin="anonymous">
9.
10.    <title>Mathifient!</title>
11.  </head>
12.  <body>
13.    <noscript>
14.      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't
        work properly without
15.      JavaScript enabled. Please enable it to continue.</strong>
16.    </noscript>
17.    <div id="app"></div>
18.    <!-- built files will be auto injected -->
19.    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0-beta1/dist/js/boot  44
        strap.bundle.min.js" integrity="sha384-pprn3073KE6t16bjs2QrFaJGz5/SUs  44
        LqktiwsUTF55Jfv3qYSDhgCecCxMW52nD2" crossorigin="anonymous"></script>
20.  </body>
21. </html>
```

This is the HTML code that should be in `public/index.html`. It has the Bootstrap CSS and JavaScript added.

EVALUATION COPY: Not to be used in class.



2.3. Using Components Inside Components

Each Vue component exports an object, known as the *options* object, because it holds the options for working with the component. This allows other Vue code to import the component. A simple options object is shown below:

SimpleComp.vue

```
<script>
export default {
  name: 'SimpleComp',
  components: {
  }
}
</script>
```

This SimpleComp component can be imported into another component within the same directory using the following code:

```
import SimpleComp from './SimpleComp.vue'
```

And then it can be used in that component's template like this:

```
<template>
  <SimpleComp />
</template>
```

This method of abstracting the parts of a user interface into reusable and self-contained components reduces the complexity of building dynamic user interfaces. For example, the user interface of a shopping cart in an e-commerce application can be very complex, but broken into its basic components, the template for rendering a shopping cart might look something like this:

```
<template>
  <ListOfProducts />
  <TotalPrice />
  <ShippingOptions />
</template>
```

The name Option

The name option, while only required for components that call themselves recursively, should generally be included as it makes debugging easier. For more information, see <https://vuejs.org/api/options-misc.html#name>.

Exercise 5: Breaking an App into Components

🕒 30 to 45 minutes

In this exercise, you will break up the user interface of the Mathificent game into subcomponents. Although it is possible to write an entire Vue application in a single component, it is generally better to break the user interface into components that can be reused. So, let's make some components!

1. Create a new file in the `mathificent/src/components` directory named `HeaderNavBar.vue`. Note that the name of this file starts with a capital letter. In Vue, the names of components always start with a capital letter. Notice also that the component name contains multiple words. Using multi-word component names for all of your components except the `App` component makes your component names more descriptive and also prevents conflicts with HTML elements that have the same name.
2. Inside this new file, add a `template` block and a `script` block.
3. Inside the `script` block, write the `export` statement:

```
export default {  
  name: 'HeaderNavBar'  
}
```

4. Cut the header element from `App.vue` and paste it inside the `template` element of `HeaderNavBar.vue`.
5. Inside the `template` in `App.vue`, add a self-closing `<HeaderNavBar />` tag right before the `h1` element where the `<header>` tag previously was:

```
<template>  
  <HeaderNavBar />  
  <h1>Mathificent</h1>  
  <footer class="navbar fixed-bottom">  
    <div class="container-fluid">  
      <a href="https://www.webucator.com" class="text-light nav-link">  
        Copyright &copy; {{new Date().getFullYear()}} Webucator  
      </a>  
    </div>  
  </footer>  
</template>
```

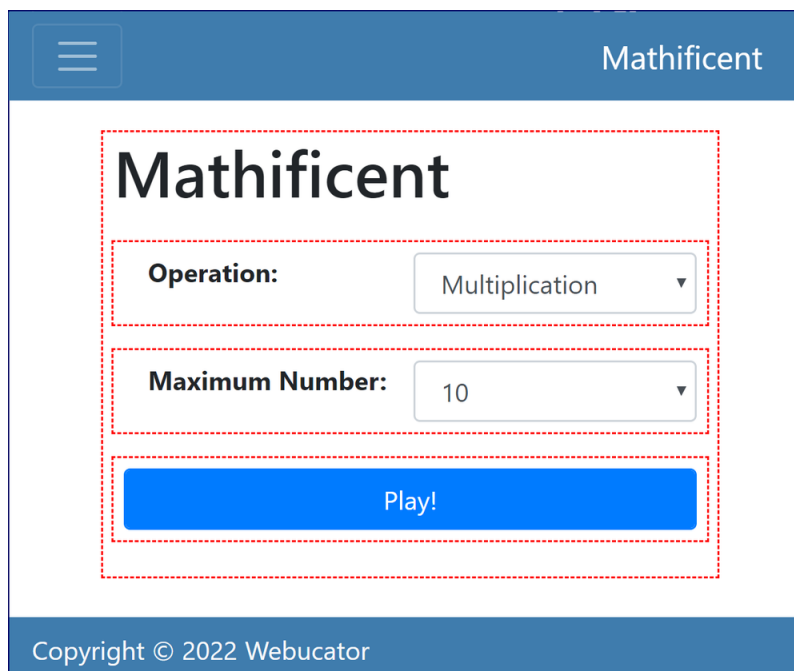
6. In the `export` statement in `App.vue`, add the `HeaderNavBar` component to the `components` property. This indicates that the `HeaderNavBar` component is a dependency of the `App` component:

```
export default {  
  name: 'app',  
  components: {  
    HeaderNavBar  
  }  
}
```

7. Use an `import` statement inside the `script` block above the `export` statement to import `HeaderNavBar` into `App`.

```
import HeaderNavBar from './components/HeaderNavBar';
```

8. Notice that when we import the `HeaderNavBar` component, we need to use `./` before the path and we don't type the `.vue` at the end of the file name.
9. Now, follow this same process to create and use a `FooterBar` component. When you're done, the app should appear just as it did before.
10. Take another look at the finished Mathificent program and think about how you might split up the main content of the app into components. Here's one way it could be done:




11. The first step in developing a user interface with components in Vue is to create a static version (meaning without any functionality) of the app. You will do this now. Create a new single-file component for each of the unique components in the following outline that you haven't already created:

- App
 - HeaderNavBar
 - *MainContainer*
 - *SelectInput*
 - *PlayButton*
 - FooterBar

12. In the `template` for each component, put a placeholder element containing the name of the component for now. For example, here's what the `SelectInput` component should look like:

```
<template>
  <div>SelectInput Component</div>
</template>

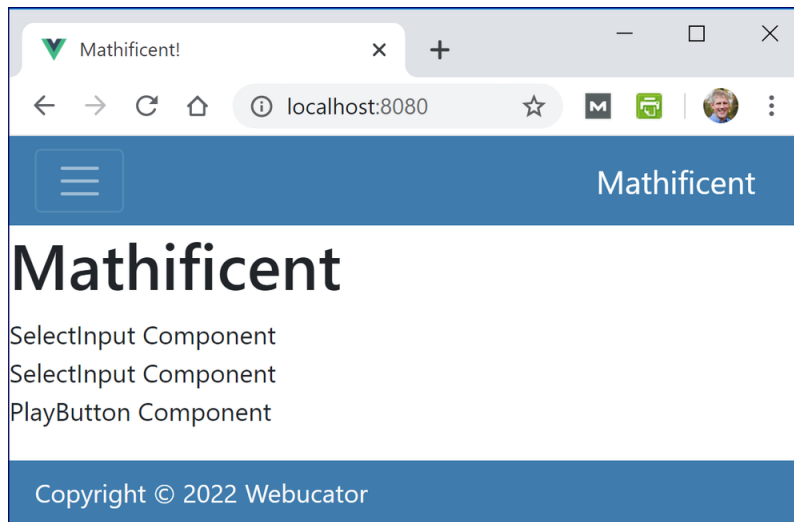
<script>
  export default {
    name: 'SelectInput'
  }
</script>
```



13. Now that you have all the components, it is time to put them together in the right order. Think about the hierarchy of components in your app:
- A. App contains HeaderNavBar, MainContainer, and FooterBar.
 - B. MainContainer contains the “Mathificent” h1 element (moved from App.vue), two instances of SelectInput, and one PlayButton.
14. Import the correct components into App.vue and MainContainer.vue and then modify the `export` statements of these two components to include the correct sub-components. Remember that, to output a child component in the `template` block, the parent component must:
- A. Import the component.

- B. Include the component in its components property.

The `MainContainer` component's template should contain a `main` element rather than a `div` element. When you're done, it should look like this in your browser:



15. In the `PlayButton` component, replace the `div` element with a `button` using this code, which uses a couple of Bootstrap classes for styling:

```
<button class="btn btn-primary">Play!</button>
```

16. In the `SelectInput` component, code the `select` dropdowns using static options and labels for now. We'll make them dynamic shortly:

```
<label for="select">Select Label</label>
<select id="select">
  <option value="sample value">Sample Value</option>
</select>
```

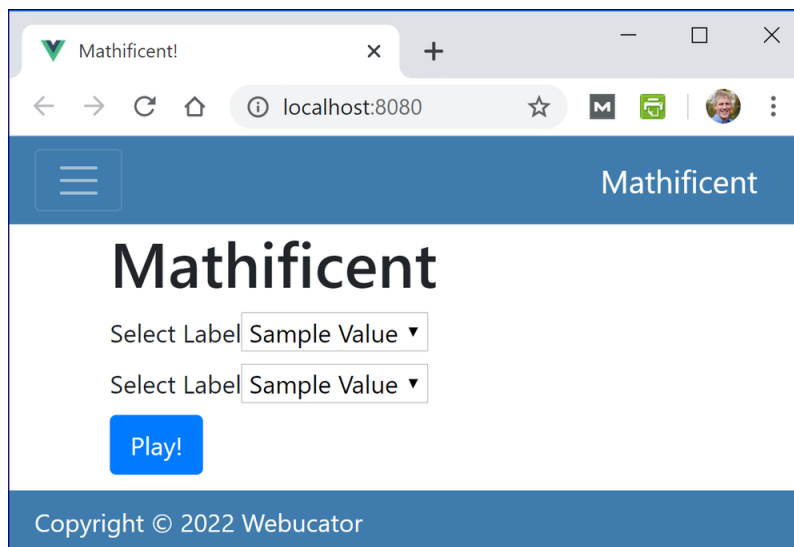
17. In `MainContainer.vue`, add an id of “main-container” to the main element:

```
<template>
  <main id="main-container">
    <h1>Mathificent</h1>
    <SelectInput />
    <SelectInput />
    <PlayButton />
  </main>
</template>
```

18. Add a style block to `MainContainer.vue` and add a rule to set the element’s width and give it some margin:

```
<style>
#main-container {
  margin: auto;
  width: 380px;
}
</style>
```

19. If it is not already running, start up your development server by running `npm run serve` from the `mathificent` directory in your terminal. Your application should now look like this:



Stop the app (**CTRL+C**) and close the terminal when you are done.

Solution:

Vue/Solutions/basic-vue-features/breaking-into-components/App.vue

```
1.   <template>
2.     <div>
3.       <HeaderNavBar />
4.       <MainContainer />
5.       <FooterBar />
6.     </div>
7.   </template>
8.
9.   <script>
10.    import HeaderNavBar from './components/HeaderNavBar';
11.    import MainContainer from './components/MainContainer';
12.    import FooterBar from './components/FooterBar';
13.
14.    export default {
15.      name: 'app',
16.      components: {
17.        HeaderNavBar,
18.        MainContainer,
19.        FooterBar
20.      }
21.    }
22.   </script>
23.
24.   <style>
25.     footer,
26.     header {
27.       background-color: #3f7cad;
28.     }
29.   </style>
```

Evaluation
Copy

Solution:

Vue/Solutions/basic-vue-features/breaking-into-components/HeaderNavBar.vue

```
1.   <template>
2.     <header>
3.       <nav class="navbar navbar-expand-lg navbar-dark">
4.         <div class="container-fluid">
5.           <button class="navbar-toggler" type="button"
6.             data-bs-toggle="collapse" data-bs-target="#navbarText">
7.             <span class="navbar-toggler-icon"></span>
8.           </button>
9.           <div class="collapse navbar-collapse" id="navbarText">
10.            <ul class="navbar-nav mr-auto text-left">
11.              <li class="nav-item active">
12.                <a class="nav-link" href="/">Home</a>
13.              </li>
14.            </ul>
15.          </div>
16.          <a class="navbar-brand" href="/">Mathificent</a>
17.        </div>
18.      </nav>
19.    </header>
20.  </template>
21.
22.  <script>
23.    export default {
24.      name: 'HeaderNavBar'
25.    }
26.  </script>
```

Solution:

Vue/Solutions/basic-vue-features/breaking-into-components/FooterBar.vue

```
1. <template>
2.   <footer class="navbar fixed-bottom">
3.     <div class="container-fluid">
4.       <a href="https://www.webucator.com" class="text-light nav-link">
5.         Copyright &copy; {{new Date().getFullYear()}} Webucator
6.       </a>
7.     </div>
8.   </footer>
9. </template>
10.
11. <script>
12.   export default {
13.     name: 'FooterBar'
14.   }
15. </script>
```

Evaluation
Copy

Solution:

Vue/Solutions/basic-vue-features/breaking-into-components/MainContainer.vue

```
1.   <template>
2.     <main id="main-container">
3.       <h1>Mathifificent</h1>
4.       <SelectInput />
5.       <SelectInput />
6.       <PlayButton />
7.     </main>
8.   </template>
9.
10.  <script>
11.    import SelectInput from './SelectInput';
12.    import PlayButton from './PlayButton';
13.
14.    export default {
15.      name: 'MainContainer',
16.      components: {
17.        SelectInput,
18.        PlayButton
19.      }
20.    }
21.  </script>
22.
23.  <style scoped>
24.    #main-container {
25.      margin: auto;
26.      width: 380px;
27.    }
28.  </style>
```

Solution:

Vue/Solutions/basic-vue-features/breaking-into-components/SelectInput.vue

```
1. <template>
2. <div>
3.   <label for="select">Select Label</label>
4.   <select id="select">
5.     <option value="sample value">Sample Value</option>
6.   </select>
7. </div>
8. </template>
9.
10. <script>
11.   export default {
12.     name: 'SelectInput'
13.   }
14. </script>
```

Solution:

Vue/Solutions/basic-vue-features/breaking-into-components/PlayButton.vue

```
1. <template>
2.   <button class="btn btn-primary">Play!</button>
3. </template>
4.
5. <script>
6.   export default {
7.     name: 'PlayButton'
8.   }
9. </script>
```

EVALUATION COPY: Not to be used in class.



2.4. Passing Data to Child Components

The components that you include inside of another component's template are known as *child* components. The component that includes the child components in its own template is called the *parent* component. Parent components can pass data to child components as *props* (short for

“properties”). To pass data as a prop, add an attribute to the tag for the custom component in the parent’s template and assign the attribute a value.

For example, if you have a custom component that reverses the letters in a string, you might use that component several times in a parent component, each time passing in a different name as a prop:

```
<ReverseString stringToReverse="Chris" />
<ReverseString stringToReverse="Nat" />
```

However, simply passing a prop to a child won’t make it usable in the child. You also need to add the prop to the props object of the child. The props object contains the props (and their data types) that can be passed into the component:

```
{
  props: {
    stringToReverse: String
  },
}
```

Once you’ve listed a prop in the props object, you can use that prop in the component.

Here is a working ReverseString component:

Demo 2.3:

Vue/demo-viewer/src/components/basic-vue-features/ReverseString.vue

```
1. <template>
2.   <h2>{{stringToReverse.split('').reverse().join('')}}</h2>
3. </template>
4.
5. <script>
6.   export default {
7.     name: "ReverseString",
8.     props: {
9.       stringToReverse: String
10.    }
11.  }
12. </script>
```

Again, this component can be included in a parent component’s template like this:

```
<ReverseString stringToReverse="Chris" />
```

Run `npm run serve` from the `demo-viewer` directory and then open `http://localhost:8080` in your browser. Notice that there are two links to reverse strings. They pass different values to the `ReverseString` component. Click them to see the strings reversed.

❖ 2.4.1. Data Types

When you pass a value with the wrong data type in as a prop, Vue does not error. Instead, it outputs a warning, which you can see in the JavaScript console. The warning will read something like:

```
[Vue warn]: Invalid prop: type check failed for prop "num". Expected Number with value 5, got String with value "5".
```

For example, consider the following component:

Demo 2.4:

Vue/demo-viewer/src/components/basic-vue-features/SquareNum.vue

```
1. <template>
2.   <h2>Square of {{num}}</h2>
3.   <p>{{num}}<sup>2</sup> = {{num * num}}</p>
4. </template>
5.
6. <script>
7.   export default {
8.     name: "SquareNum",
9.     props: {
10.       num: Number
11.     }
12.   }
13. </script>
-----Lines 14 through 23 Omitted-----
```

If you include this component with the following tag, a string will be passed in for the `num` value:

```
<SquareNum num="5" />
```

Vue will generate the warning shown above.

However, by default, all prop values passed in as attributes in this way will be strings. We'll learn how to pass in different data types soon.

EVALUATION COPY: Not to be used in class.



2.5. Dynamic Data in Templates

At its most basic level, the job of a Vue component is to merge dynamic data with template code to produce output. Vue provides several ways for generating and working with this dynamic data and for including it in templates:

1. Computed properties.
2. The data object.
3. The methods object.

We'll look at each of these now.

Evaluation
Copy

EVALUATION COPY: Not to be used in class.



2.6. Computed Properties

As you saw in the previous example, JavaScript code within double curly braces in a template will get interpreted; however, it's generally not considered a good practice to muddle up your template with a lot of calculations and complex JavaScript. Instead, any time you have a piece of dynamic data that can be calculated based on other data in the template, you should consider making it into a *computed property*.

Computed properties are functions that are written as properties of the `computed` object in the options object of a Vue instance. For example, you can turn our message reversal code into a computed property like this:


```
{
  computed: {
    reversedString: function() {
      return this.stringToReverse.split('').reverse().join('');
    }
  }
}
```

With this computed property function written, you can now use the computed property inside your template like you would use any other property:

```
<template>
  <div>
    Your name in reverse is {{reversedString}}.
  </div>
</template>
```

Here is a working ReverseStringComputed component:

Demo 2.5:

Vue/demo-viewer/src/components/basic-vue-features/ReverseStringComputed.vue

```
1.  <template>
2.    <h2>{{reversedString}}</h2>
3.  </template>
4.
5.  <script>
6.    export default {
7.      name: "ReverseStringComputed",
8.      props: {
9.        stringToReverse: String
10.     },
11.     computed: {
12.       reversedString: function() {
13.         return this.stringToReverse.split('').reverse().join('');
14.       }
15.     }
16.   }
17. </script>
```

Besides being useful for neatening up your template, computed properties have another superpower: they're cached. If the data that goes into calculating a computed property doesn't change, then there's

no reason for the computed property to be updated when a template is rendered, so Vue will use the cached value of the computed property instead. This caching can make your application run faster by sparing the user's computer from having to do unnecessary calculations.

EVALUATION COPY: Not to be used in class.



2.7. The data Object

Vue's `data` object contains the properties that control when the view updates. Because Vue monitors them and reacts to changes in them, these properties are called “reactive data.”

The `data` property of a component holds a function that returns the component's state. By returning a function, rather than the data itself, each instance of a component within an application can have its own unique state. To understand this better, consider the following:

```
const a = function() {  
  return ['a', 'b'];  
};  
  
const foo = a();  
const bar = a();
```

**Evaluation
Copy**

Will `foo === bar` be true or false? Even though the two arrays hold the same content, it will be false, because the function creates and returns a new instance of the array every time it is called. As such, we can change the `bar` array without it affecting the `foo` array. For a component that is only used once in a Vue application, this wouldn't make any difference, but for components that are used more than once, it is essential that each has its own unique state.

2.7. The methods Object

The `methods` object contains the functions that modify properties in the `data` object. Unlike computed properties, the return values of methods are never cached and the function will do its job every time it is called. For this reason, you should use a computed property if the value returned is likely to be relatively static.

Consider the following component:

Demo 2.6:

Vue/demo-viewer/src/components/basic-vue-features/SpellWord.vue

```
1.   <template>
2.     <div>
3.       <p>{{wordToSpell}}</p>
4.     </div>
5.   </template>
6.
7.   <script>
8.     export default {
9.       name: "SpellWord",
10.      props: {
11.        word: String
12.      },
13.      data: function() {
14.        return {
15.          index: 0,
16.          wordToSpell: ''
17.        }
18.      },
19.      methods: {
20.        spell() {
21.          if (this.index < this.word.length) {
22.            this.index++;
23.            this.wordToSpell = this.word.toLowerCase().substring(0, this.index);
24.            setTimeout(this.spell, 500);
25.          } else {
26.            this.wordToSpell = this.word.toUpperCase();
27.          }
28.        }
29.      },
30.      created: function() {
31.        this.spell();
32.      }
33.    }
34.  </script>
```

Things to note about the SpellWord component:

1. It gets passed a word prop.
2. It has two data properties: index and wordToSpell, which default to 0 and ' ', respectively.

3. It has a `spell()` method which sets `this.wordToSpell` to a longer substring of `word` every half second until the complete word is spelled out, at which point, it sets `this.wordToSpell` to the full value of `word` in uppercase letters.

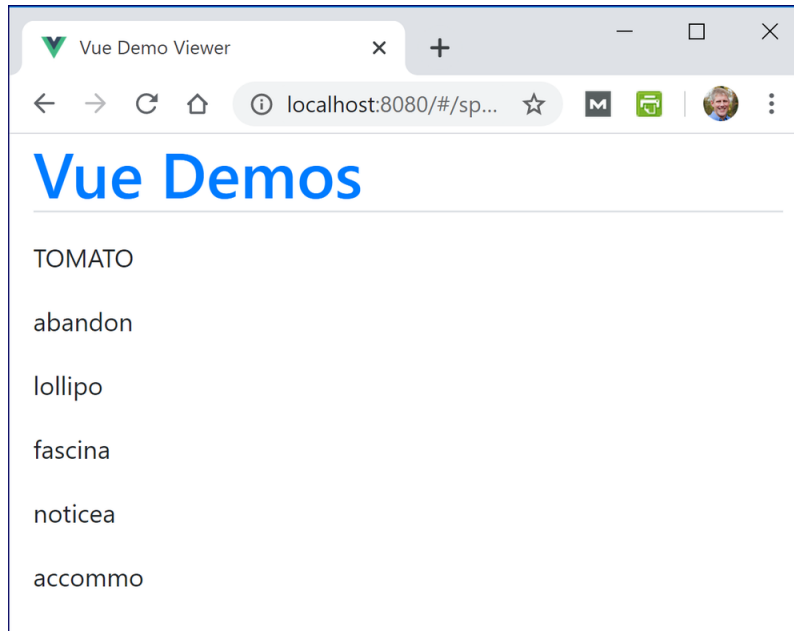
If the demo-viewer app isn't already running, run `npm run serve` from the `demo-viewer` directory and then open `http://localhost:8080` in your browser. Then click the [Spell Words](#) link under **Basic Vue Features**. You should see a page that spells out several words. The `SpellWords` component looks like this:

Demo 2.7:

Vue/demo-viewer/src/components/basic-vue-features/SpellWords.vue

```
1. <template>
2.   <SpellWord word="Tomato" />
3.   <SpellWord word="Abandon" />
4.   <SpellWord word="Lollipop" />
5.   <SpellWord word="Fascinate" />
6.   <SpellWord word="Noticeable" />
7.   <SpellWord word="Accommodate" />
8. </template>
9.
10. <script>
11. import SpellWord from './SpellWord.vue';
12.
13. export default {
14.   name: "SpellWords",
15.   components: {
16.     SpellWord
17.   }
18. }
19. </script>
```

Here is the page midway through the spellings:



The most important thing to notice is that each `SpellWord` component has its own data. As explained earlier, this is because the `SpellWord` component's data object returns a function and not a static object.

The `this` Keyword

Take another look at the `spell()` method from the `SpellWord` component:

```
spell() {  
  if (this.index < this.word.length) {  
    this.index++;  
    this.wordToSpell = this.word.toLowerCase().substring(0, this.index);  
    setTimeout(this.spell, 500);  
  } else {  
    this.wordToSpell = this.word.toUpperCase();  
  }  
}
```

What is `this`? The `this` object is a special object in JavaScript that refers to the current object. In Vue components, `this` refers to the component itself. From a component's methods, you must use `this` to access the component's props, data properties, and methods themselves. If you fail to use `this`, the method will look for a local variable (one defined within the method itself) instead of an instance variable (one defined as a property of the component).

2.7. Instance Lifecycle Hooks

At the bottom of the `SpellWord` component, you may have noticed this code:

```
created: function() {  
  this.spell();  
}
```

The `created` property is an *instance lifecycle hook*. It runs immediately after an instance is created. We use it to call `this.spell()`, which then recursively calls itself every half second.

Conclusion

In this lesson, you have learned to change the data in Vue components and to dynamically update their templates.

LESSON 3

Directives

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ Vue 3 directives.
- ☒ Conditional rendering with `v-if`, `v-else-if`, and `v-else`.
- ☒ Binding HTML elements to fields with `v-model` and `v-bind`.
- ☒ Creating event listeners with `v-on`.
- ☒ Looping with `v-for`.
- ☒ Emitting events from child components.
- ☒ Listening for events in parent components.
- ☒ Passing data in event emitters.

Introduction

Directives are special attributes that take JavaScript expressions as values. They make it possible to do many of the most common operations in a component directly in the template. In this lesson, in addition to learning about directives, you will learn to emit custom events and listen for those events in parent components.

EVALUATION COPY: Not to be used in class.



3.1. Directives

Vue's built-in directives are prefixed with “v-”. For example, the `v-show` directive takes a JavaScript statement that evaluates to `true` or `false` and toggles the value of the element's `display` property based on the result of the statement:

```
<p v-show="loggedIn">You are logged in as {{yourName}}.</p>
```

In the above example, the paragraph containing the “logged in” message will only display if the variable named `loggedIn` evaluates to `true`.

Here is another example:

Demo 3.1: `Vue/demo-viewer/src/components/directives/VShow.vue`

```
1. <template>
2.   <h3>{{seconds}}</h3>
3.   <p v-show="seconds % 2 === 0">I love chocolate!</p>
4.   <p v-show="seconds % 2 === 1">I love vanilla!</p>
5. </template>
6.
7. <script>
8.   export default {
9.     name: "VShow",
10.    data: function() {
11.      return {
12.        seconds: new Date().getSeconds()
13.      }
14.    }
15.  }
16. </script>
```

If the demo-viewer app isn't already running, run `npm run serve` from the `demo-viewer` directory and then open `http://localhost:8080` in your browser. Then click the **v-show** link under **Directives**. If the current value of seconds on your computer's clock is even, you will get a message saying “I love chocolate!” Otherwise, you'll get a message saying “I love vanilla.” You can run the component again by refreshing the page.

The most common uses for directives are:

1. Conditional rendering with `v-if`, `v-else-if`, and `v-else`.

2. Binding HTML elements to fields with `v-model` and `v-bind`.
3. Creating event listeners with `v-on`.
4. Looping with `v-for`.

We'll be using all of these directives in our Mathificent app, so let's learn how they all work.

EVALUATION COPY: Not to be used in class.



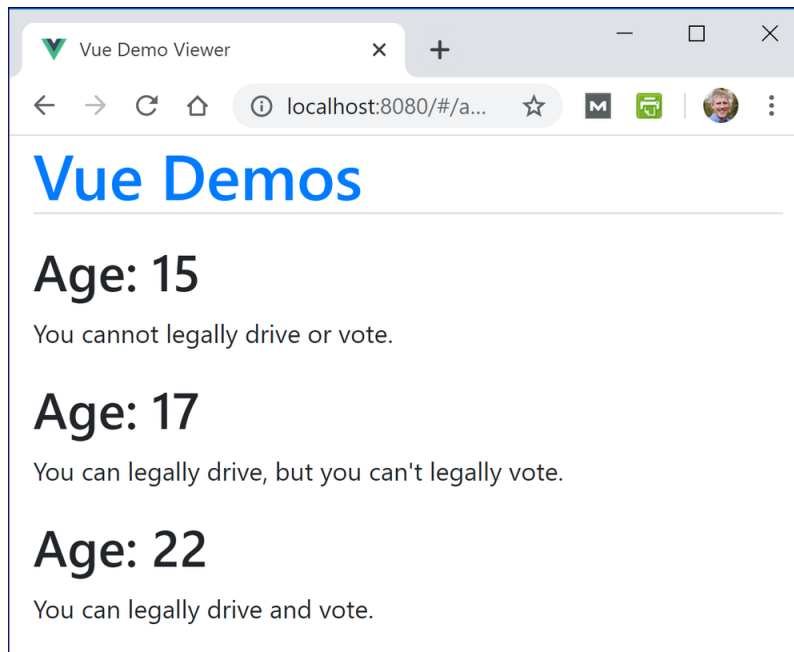
3.2. Conditionals with `v-if` / `v-else-if` / `v-else`

The `v-if`, `v-else-if`, and `v-else` directives work just like `if`, `else if`, and `else` in JavaScript:

Demo 3.2: `Vue/demo-viewer/src/components/directives/AgeComp.vue`

```
1. <template>
2.   <h2>Age: {{age}}</h2>
3.   <div v-if="age >= 18">
4.     <p>You can legally drive and vote.</p>
5.   </div>
6.   <div v-else-if="age >= 16">
7.     <p>You can legally drive, but you can't legally vote.</p>
8.   </div>
9.   <div v-else>
10.    <p>You cannot legally drive or vote.</p>
11.  </div>
12. </template>
13.
14. <script>
15.   export default {
16.     name: "AgeComp",
17.     props: {
18.       age: Number
19.     }
20.   }
21. </script>
-----Lines 22 through 27 Omitted-----
```

On the home page of the demo-viewer app, click the [Ages](#) link under **Directives**. This runs the AgesComp component, which outputs three AgeComp components passing in 15, 17, and 22. The message output is dependent on the value of the passed-in age prop:



As we saw earlier, the `v-show` directive can also be used for conditional rendering. The difference between using `v-if` and `v-show` is that `v-show` renders an element into the DOM and then determines whether or not to show it, using the CSS `display` property. The `v-if` directive evaluates its expression and determines whether or not to render an element into the DOM based on the result of the expression.

EVALUATION COPY: Not to be used in class.



3.3. Two-way Binding with `v-model`

Two-way data binding in Vue makes working with forms and data simple. Here's an example:

```
<h3>{{message}}</h3>
<input v-model="message">
```

Two-way data binding creates a connection between the `input` element and a data property named “message” so that when the user enters data into the form input, it changes the value of the data property, and when the data property changes, it updates the value of the form input.

The `v-model` directive can also be used with other types of form fields:

Textareas

Binding textareas works the same way as binding text inputs:

```
<textarea v-model="comments"></textarea>
```

Checkboxes

To bind a checkbox, pass a Boolean value into the `v-model` directive. In the following example, `isChecked` will contain `true` or `false` depending on whether or not the checkbox is checked:

```
<h3>{{isChecked}}</h3>
<input type="checkbox" v-model="isChecked">
```

Also, because it is *two-way* binding, if you set the value of `isChecked` to `true` in the data object, the checkbox will be checked by default.

Radio Buttons

For radio buttons, the value of the selected radio button will be used to set the value of the bound `fruit` property:

```
<input name="fruit" value="banana" type="radio" v-model="fruit"> Banana
<input name="fruit" value="apple" type="radio" v-model="fruit"> Apple
<input name="fruit" value="pear" type="radio" v-model="fruit"> Pear
```

Again, because it is two-way binding, you can preselect a radio button by setting the value of `fruit` in the data object.

Select Menus

Select menus work just like radio buttons. The `v-model` attribute goes in the `<select>` tag:

```
<select name="veggie" v-model="veggie">
  <option disabled value="">Select a veggie</option>
  <option value="eggplant">Eggplant</option>
  <option value="squash">Squash</option>
  <option value="zucchini">Zucchini</option>
</select>
```

And again, because it is two-way binding, you can preselect an option by setting the value of `veggie` in the data object.

v-model Examples

Try out these `v-model` examples in the demo-viewer app. If it isn't already running, run `npm run serve` from the `demo-viewer` directory and then open `http://localhost:8080` in your browser. Then click the [v-model](#) link under **Directives**.

Here is the component code:

Demo 3.3: Vue/demo-viewer/src/components/directives/VModel.vue

```
1.  <template>
2.    <h2>Text Input and Textarea</h2>
3.    <h3>{{message}}</h3>
4.    <input v-model="message" class="form-control">
5.
6.    <h3>{{comments}}</h3>
7.    <textarea v-model="comments" class="form-control"></textarea>
8.
9.    <h2>Checkbox</h2>
10.   <h3>{{isChecked}}</h3>
11.   <input type="checkbox" v-model="isChecked">
12.
13.   <h2>Radio Buttons</h2>
14.   <h3>{{fruit}}</h3>
15.   <input name="fruit" value="banana" type="radio" v-model="fruit"> Banana
16.   <input name="fruit" value="apple" type="radio" v-model="fruit"> Apple
17.   <input name="fruit" value="pear" type="radio" v-model="fruit"> Pear
18.
19.   <h2>Select Menu</h2>
20.   <h3>{{veggie}}</h3>
21.   <select name="veggie" v-model="veggie">
22.     <option disabled value="">Select a veggie</option>
23.     <option value="eggplant">Eggplant</option>
24.     <option value="squash">Squash</option>
25.     <option value="zucchini">Zucchini</option>
26.   </select>
27. </template>
28.
29. <script>
30. export default {
31.   name: "VModel",
32.   data: function() {
33.     return {
34.       message: 'Change me.',
35.       comments: '',
36.       isChecked: true,
37.       fruit: 'pear',
38.       veggie: 'squash'
39.     }
40.   }
41. }
42.
43. -----Lines 41 through 61 Omitted-----
```



3.4. One-way Data Binding, Repeating, and Event Handling

Vue can also do one-way data binding with the `v-bind` directive. The `v-bind` directive creates a one-way link from a dynamic property to any attribute you specify. To bind an attribute, prefix the attribute with `v-bind::`:

```
<button v-bind:disabled="isDisabled">Click me!</button>
```

When the value of `isDisabled` changes, the button's `disabled` property changes along with it.

❖ 3.4.1. `v-bind` Shorthand

The `v-bind` directive is one of the most frequently used Vue directives. To make using it even simpler, Vue contains a shorthand method for writing it. Instead of writing out the full `v-bind` directive followed by a colon and the attribute name, you can just use a colon before an attribute name, like this:

```
<button :disabled="isDisabled">Click me!</button>
```

Passing Non-String Values as Props

Earlier (see page 43), we showed the warning that you get when you pass a component a prop of the wrong data type. To pass in a non-string prop, use `v-bind` like this:

```
<SquareNum v-bind:numToSquare="5" />
```

Or the shorthand version:

```
<SquareNum :numToSquare="5" />
```

This tells Vue that the value of the `numToSquare` attribute is JavaScript and not an HTML string.

3.4. Repeating an Element using v-for

The `v-for` directive makes it easy to output multiple elements or components of the same type. Assume you have the following array of objects:

```
presidents: [  
  {  
    id: 1,  
    name: "Washington"  
  },  
  {  
    id: 2,  
    name: "Adams"  
  },  
  {  
    id: 3,  
    name: "Jefferson"  
  }  
]
```

The following code will create a button for each president:

```
<button v-for="president in presidents" v-bind:key="president.id">  
  {{president.name}}  
</button>
```

The value of `v-for` is simple to understand: it's just `item in items`, where `items` is an array and `item` is the variable holding the current element in the array.

Vue uses the key to track each node. The value of each key in the series must be unique. In this case, we bind it to each array element's `id`.

Remember that we can use the shorthand for `v-bind`:


```
<button v-for="president in presidents" :key="president.id">  
  {{president.name}}  
</button>
```

The following code shows the complete example:

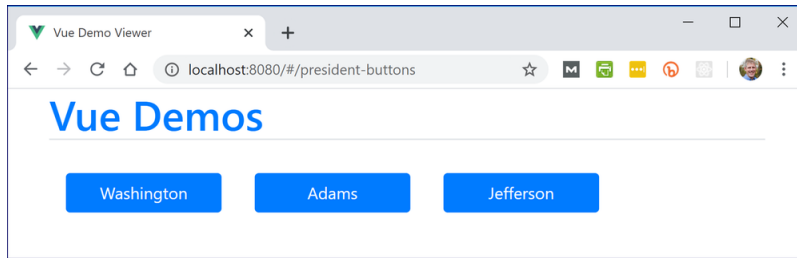
Demo 3.4:

Vue/demo-viewer/src/components/directives/PresidentButtons.vue

```
1. <template>
2.   <button class="btn btn-primary"
3.     v-for="president in presidents" :key="president.id">
4.     {{president.name}}
5.   </button>
6. </template>
7.
8. <script>
9. export default {
10.   name: "PresidentButtons",
11.   data: function() {
12.     return {
13.       presidents: [
14.         {
15.           id: 1,
16.           name: "Washington"
17.         },
18.         {
19.           id: 2,
20.           name: "Adams"
21.         },
22.         {
23.           id: 3,
24.           name: "Jefferson"
25.         }
26.       ]
27.     }
28.   },
29. }
30. </script>
-----Lines 31 through 44 Omitted-----
```



If the demo-viewer app isn't already running, run `npm run serve` from the demo-viewer directory and then open `http://localhost:8080` in your browser. Then click the **President Buttons** link under **Directives**. You should see a button for each president:

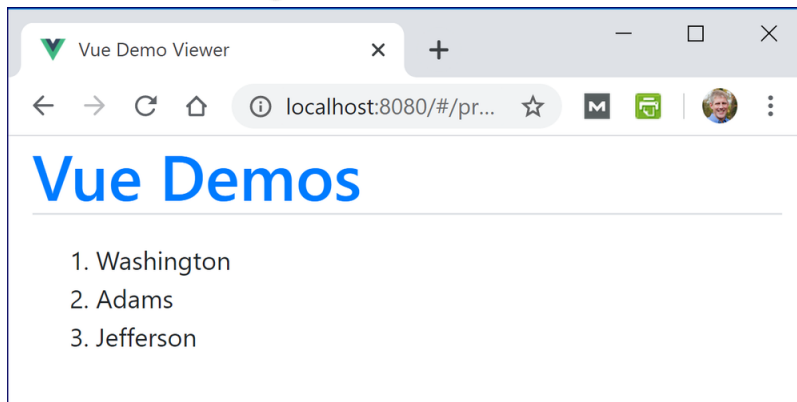


Try This

Open `Vue/demo-viewer/src/components/directives/PresidentButtons.vue` in your editor and change the template to output a list:

```
<template>
<ol>
  <li v-for="president in presidents" :key="president.id">
    {{president.name}}
  </li>
</ol>
</template>
```

The page should re-render and show a list:



Try adding another president to the array:

```
{
  id: 4,
  name: "Madison"
}
```

Another list item should appear.

3.4. Event Handling

Most HTML elements have certain events that can happen to them. For example, changing the value of a form field produces a “change” event, clicking an element produces a “click” event (for most elements), and when an element first loads into the browser DOM, it emits a “load” event.

Programmers use event listeners to listen for these events and, via *callback functions*, cause something else to happen in response.

In Vue, setting event listeners that trigger some other action can be done using the `v-on` directive. The `v-on` directive creates an event listener for the element it is a part of and it takes as its value a JavaScript statement or the name of a method (defined in the `methods` property) that should run when the event occurs.

The following code would call the `increment()` method every time the button is clicked and would set `count` to 0 when the mouse moves off of the button.

```
<button class="btn btn-primary"
  v-on:click="increment()"
  v-on:mouseout="count=0">{{count}}</button>
```

❖ 3.4.2. v-on Shorthand

The `v-on` directive, like the `v-bind` directive, is one of the most commonly-used directives. For this reason, it also has a shorthand form. Instead of writing the full `v-on` directive, you can just use the `@` symbol, followed by the event you want to listen for. For example:

```
<button class="btn btn-primary"
  @click="increment()"
  @mouseout="count=0">{{count}}</button>
```

The following code shows the complete example:

Demo 3.5:

Vue/demo-viewer/src/components/directives/CounterComp.vue

```
1. <template>
2.   <button class="btn btn-primary"
3.     @click="increment"
4.     @mouseout="count=0">{{count}}</button>
5. </template>
6.
7. <script>
8. export default {
9.   name: "CounterComp",
10.  data: function() {
11.    return {
12.      count: 0
13.    }
14.  },
15.  methods: {
16.    increment: function() {
17.      this.count++;
18.    }
19.  }
20. }
21. </script>
```

-----Lines 22 through 36 Omitted-----

If the demo-viewer app isn't already running, run `npm run serve` from the `demo-viewer` directory and then open `http://localhost:8080` in your browser. Then click the **Counter** link under **Directives**. Without moving the cursor off the button, click it again to see the counter increment. Move your mouse off the button to see it get reset to 0.

3.4. Putting it All Together

Take a look at the following code, which combines the `v-bind`, `v-for`, and `v-on` directives, and also uses the `v-html` directive to output unescaped HTML code. Note that it uses the shorthand versions of `v-bind` and `v-on`:

Demo 3.6: Vue/demo-viewer/src/components/directives/QuotesComp.vue

```
1. <template>
2.   <button class="btn btn-primary"
3.     v-for="quote in quotes" :key="quote.id"
4.     @click="currentQuoteId=quote.id"
5.     :disabled="isCurrentQuoteButton(quote)">{{quote.president}}</button>
6.   <article v-html="content"></article>
7. </template>
8.
9. <script>
10. export default {
11.   name: "QuotesComp",
12.   data: function() {
13.     return {
14.       currentQuoteId: 1,
15.       quotes: [
16.         {
17.           id: 1,
18.           president: "Washington",
19.           content: `It is <strong>infinitely</strong> better to have
20.             a few <em>good</em> men than many
21.             <em>indifferent</em> ones.
22.         },
23.         {
24.           id: 2,
25.           president: "Adams",
26.           content: `<em>If conscience disapproves</em>, the
27.             <strong>loudest</strong> applauses of the
28.             world are of little value.`
29.         },
30.         {
31.           id: 3,
32.           president: "Jefferson",
33.           content: `The most valuable of all talents is that
34.             of never using <em>two</em> words when
35.             <em>one</em> will do.`
36.         }
37.       ]
38.     },
39.     computed: {
40.       content: function() {
41.         const quote = this.quotes.find(quote => {
42.           return quote.id === this.currentQuoteId
43.         });
44.       }
45.     }
46.   }
47. }
```

```

45.         return `<q>${quote.content}</q><br/> - ${quote.president}`;
46.     }
47. },
48.     methods: {
49.         isCurrentQuoteButton(quote) {
50.             return (this.currentQuoteId === quote.id);
51.         }
52.     }
53. }
54. </script>
-----Lines 55 through 73 Omitted-----

```

Open the demo-viewer app in your browser. Then click the [Quotes](#) link under **Directives**. Click the buttons to see the presidents' quotes.

Things to notice:

1. The `v-for` directive is used to loop through the `quotes` array.
2. The shorthand version of the `v-on` directive is used to set `currentQuoteId` to `quote.id`, where `quote` is the current quote in the array loop.
3. The shorthand version of the `v-bind` directive is used to bind the `disabled` attribute to the `isCurrentQuoteButton()` method, which returns `true` if the passed-in quote is the current quote.
4. `content` is a computed property, which returns an HTML string to put into the `article` element.

EVALUATION COPY: Not to be used in class.



3.5. Emitting Custom Events

In addition to the built-in events that HTML elements emit, it's possible to emit custom events in reaction to interactions. To emit a custom event, use the `$emit` instance method. The `$emit` method takes the name of your custom event and, optionally, a value. For example, the following button emits a custom "hide-me" event and passes the string "fade" to the callback function.

```
<button id="close-button"  
  @click="$emit('hide-me', 'fade')">X</button>
```

You listen for custom events the same way that you listen for built-in events. The passed-in value, if there is one, is available through the `$event` parameter:

```
v-on:hide-me="hideChild($event)"
```

Or, using the shorthand:

```
@hide-me="hideChild($event)"
```

Take a look at the following component:

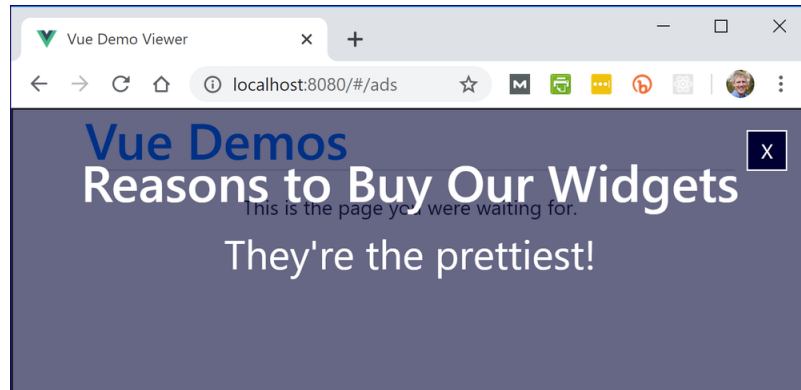
**Evaluation
Copy**

Demo 3.7: Vue/demo-viewer/src/components/directives/AdComp.vue

```
1.  <template>
2.    <div id="ad">
3.      <button id="close-button" v-show="num <= start - 3"
4.        @click="$emit('hide-me', 'fade')">X</button>
5.      <div id="ad-text">
6.        <h1>Reasons to Buy Our Widgets</h1>
7.        <p>{{reasons[reasonNum]}}</p>
8.      </div>
9.    </div>
10. </template>
11. <script>
12. export default {
13.   name: "AdComp",
14.   props: {
15.     start: Number
16.   },
17.   data: function() {
18.     return {
19.       num: this.start,
20.       reasons: [
21.         "They're the cheapest!", "They're the coolest!",
22.         "They're the hottest!", "They're the awesomest!",
23.         "They're the prettiest!", "They're the best!"
24.       ],
25.       reasonNum: 0
26.     }
27.   },
28.   methods: {
29.     countDown() {
30.       this.num--;
31.       this.reasonNum++;
32.       if (this.num < 0) {
33.         this.$emit('hide-me');
34.       } else {
35.         setTimeout(this.countDown, 1000);
36.       }
37.     }
38.   },
39.   created: function() {
40.     setTimeout(this.countDown, 1000);
41.   }
42. }
43. </script>
-----Lines 44 through 84 Omitted-----
```

Things to notice:

1. This component is meant to be imported into other components. It will create a semi-transparent ad that fully covers the viewport:



2. The `countDown()` method decrements the `num` property by 1 every second.
3. An “X” button will appear after three seconds have passed:

```
<button id="close-button" v-show="num <= start - 3"
@click="$emit('hide-me', 'fade')">X</button>
```

4. When the “X” button is clicked, it will emit a custom “hide-me” event and pass “fade” to the callback function:

```
<button id="close-button" v-show="num <= start - 3"
@click="$emit('hide-me', 'fade')">X</button>
```

5. The `countDown()` function also emits the custom “hide-me” event, but it doesn’t pass in a value. It could. We’re just demonstrating that it doesn’t have to:


```
if (this.num < 0) {
  this.$emit('hide-me');
} else {
  this.timer = setTimeout(this.countDown, 1000);
}
```

To see this working, open the demo-viewer app in your browser. Then click the [Ad Container](#) link under **Directives**.

Next look at this component, which imports and displays the `AdComp` component:

Demo 3.8: Vue/demo-viewer/src/components/directives/AdContainer.vue

```
1.  <template>
2.    <div class="container">
3.      <AdComp :start="5" @hide-me="hideMe($event)" v-show="visible"
4.        :class="className" />
5.      <p class="text-center">This is the page you were waiting for.</p>
6.    </div>
7.  </template>
8.
9.  <script>
10. import AdComp from './AdComp.vue'
11. export default {
12.   name: "AdContainer",
13.   components: {
14.     AdComp
15.   },
16.   data: function() {
17.     return {
18.       visible: true,
19.       className: ''
20.     }
21.   },
22.   methods: {
23.     hideMe(className) {
24.       if (className) {
25.         this.className = className;
26.       } else {
27.         this.visible = false;
28.       }
29.     }
30.   }
31. }
32. </script>
```



Here's the code that includes the AdComp component:

```
<AdComp :start="5" @hide-me="hideMe($event)" v-show="visible"
:class="className" />
```

Things to notice:

1. The `start` attribute uses the `v-bind` shorthand. That could also be written: `v-bind:start="5"`. This indicates that the value of the `start` attribute is not an HTML string.
2. When the `hide-me` event fires, the `hideMe()` method will be called with the `$event` argument.
3. The `v-show` directive is tied to the `visible` property. If `visible` becomes `false`, the component will be hidden.
4. The `class` attribute is bound to the `className` property.

The `hideMe()` method looks like this:

```
hideMe(className) {  
  if (className) {  
    this.className = className;  
  } else {  
    this.visible = false;  
  }  
}
```

If a value is passed in for `className`, it sets `this.className` to `className`. Remember that, when the user clicks the “X” button, we pass in the “fade” class, which is a Bootstrap class that hides an element with a fade effect.

If no value is passed in for `className`, the method sets `this.visible` to `false`, which will abruptly hide the element.

To see this working, open the demo-viewer app in your browser. Then click the [AdContainer](#) link under **Directives**.

Additional Examples

There are two additional examples that use directives and event listeners in the demo-viewer app that you might find useful to review:

1. `Vue/demo-viewer/src/components/directives/ListsComp.vue`, which has a child `ListComp` component.
2. `Vue/demo-viewer/src/components/directives/quiz/QuizComp.vue`, which imports a JSON object.

Run the components in the demo-viewer app by clicking the [Lists](#) and [Quiz](#) links under Directives. Then review the code and make sure you understand how they work. Feel free to play around with the code.

Remember to stop the app (**CTRL+C**) and close the terminal when you are done.

Conclusion

In this lesson, you have learned to work with Vue 3's directives. In the next lesson, you will use these directives to build out the Mathificent game.

LESSON 4

Implementing Game Logic

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ Working with your new Vue skills.

Introduction

In this lesson, you will build out most of the Mathificent game in a series of exercises.

Exercise 6: Passing Data Between Components

⌚ 20 to 30 minutes

In this exercise, you begin passing data between components.

1. In the `export` statement for `MainContainer.vue` (from the `Exercises/mathificient/src/components` directory), after the `components` property, add a `data` function that returns an `operations` array for the **Operations** dropdown:

```
data: function() {  
  return {  
    operations: [  
      ['Addition', '+'],  
      ['Subtraction', '-'],  
      ['Multiplication', 'x'],  
      ['Division', '/']  
    ],  
  }  
}
```

Evaluation
Copy

Notice that each element of the `operations` array is a 2-element array containing the *text* (the operation name) and the *value* (the operation symbol) that will go in a select option.

2. Add a computed property to make an array of numbers for the **Maximum Number** dropdown.

```
computed: {  
  numbers: function() {  
    const numbers = [];  
    for (let number = 2; number <= 100; number++) {  
      numbers.push([number, number]);  
    }  
    return numbers;  
  }  
}
```

Here, we're creating an array of numbers that we'll use to populate the **Maximum Number** select input. As with the `operations` array, each element of the `numbers` array is a 2-element array, containing the *value* and the *text* of the option in the select dropdown. In this case, the value and the text are the same: the number.

3. In the `MainContainer` component's template, pass the properties you just created in `MainContainer.vue` to the `SelectInput` components.

```
<SelectInput label="Operation" id="operation" :options="operations" />
<SelectInput label="Maximum Number" id="max-number" :options="numbers" />
```

4. In the export for `SelectInput`, specify the props that will be passed into the component:

```
export default {
  name: 'SelectInput',
  props: {
    id: String,
    label: String,
    options: Array
  }
}
```

5. Use the value of the `label` prop as the `label` for the the select:

```
<label>{{label}}</label>...
```

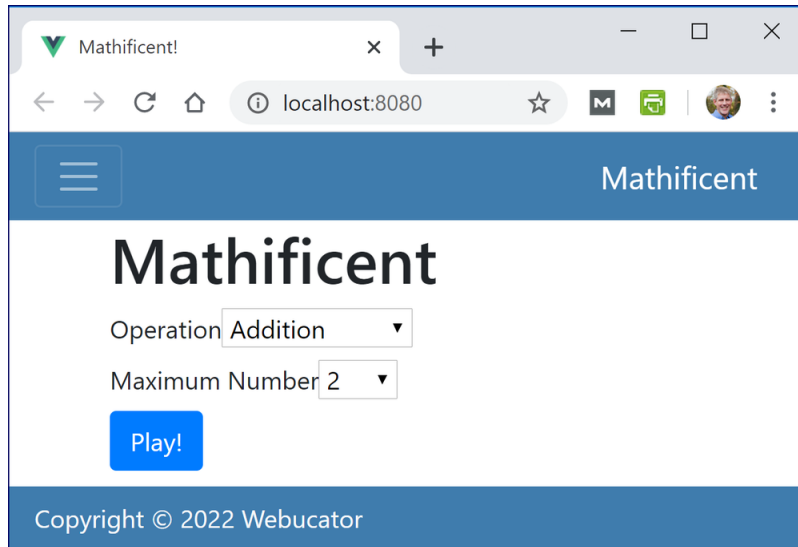
6. Bind the values of the `id` attribute of the `select` element and the `for` value of the `label` element to the `id` prop:

```
<label :for="id">{{label}}</label>
<select :id="id">
```

7. Use `v-for` to loop over the elements in the `numbers` array to make multiple option elements:

```
<option v-for="option in options" :key="option[1]"
  :value="option[1]">{{option[0]}}</option>
```

8. Start up your development server by running `npm run serve` from the `mathificent` directory in your terminal. Your application should now look like this:



Leave the app running for the remainder of this lesson.

EVALUATION COPY: Not to be used in class.

Solution:

Vue/Solutions/implementing-game/passing-data/MainContainer.vue

```
1.   <template>
2.     <main id="main-container">
3.       <h1>Mathificent</h1>
4.       <SelectInput label="Operation" id="operation" :options="operations" />
5.       <SelectInput label="Maximum Number" id="max-number" :options="numbers" />
6.       <PlayButton />
7.     </main>
8.   </template>
9.
10.  <script>
11.    import SelectInput from './SelectInput';
12.    import PlayButton from './PlayButton';
13.
14.    export default {
15.      name: 'MainContainer',
16.      components: {
17.        SelectInput,
18.        PlayButton
19.      },
20.      data: function() {
21.        return {
22.          operations: [
23.            ['Addition', '+'],
24.            ['Subtraction', '-'],
25.            ['Multiplication', 'x'],
26.            ['Division', '/']
27.          ],
28.        }
29.      },
30.      computed: {
31.        numbers: function() {
32.          const numbers = [];
33.          for (let number = 2; number <= 100; number++) {
34.            numbers.push([number, number]);
35.          }
36.          return numbers;
37.        }
38.      }
39.    }
40.  </script>
    -----Lines 41 through 47 Omitted-----
```

Solution:

Vue/Solutions/implementing-game/passing-data/SelectInput.vue

```
1.   <template>
2.     <div>
3.       <label :for="id">{{label}}</label>
4.       <select :id="id">
5.         <option v-for="option in options" :key="option[1]"
6.           :value="option[1]">
7.           {{option[0]}}
8.         </option>
9.       </select>
10.    </div>
11.  </template>
12.
13.  <script>
14.    export default {
15.      name: 'SelectInput',
16.      props: {
17.        id: String,
18.        label: String,
19.        options: Array
20.      }
21.    }
22.  </script>
```

Evaluation
Copy



Exercise 7: Vue Data Binding

⌚ 25 to 40 minutes

In this exercise, you will bind data to input elements.

1. Add a `v-model` directive to the `select` element in the `SelectInput` component:

```
<select :id="id" v-model="currentValue">
```

2. To the component's export statement, add a `data` property that holds a function that returns `currentValue` with a default of an empty string:

```
export default {  
  name: 'SelectInput',  
  props: {  
    id: String,  
    label: String,  
    options: Array  
  },  
  data: function() {  
    return {  
      currentValue: ''  
    }  
  }  
}
```

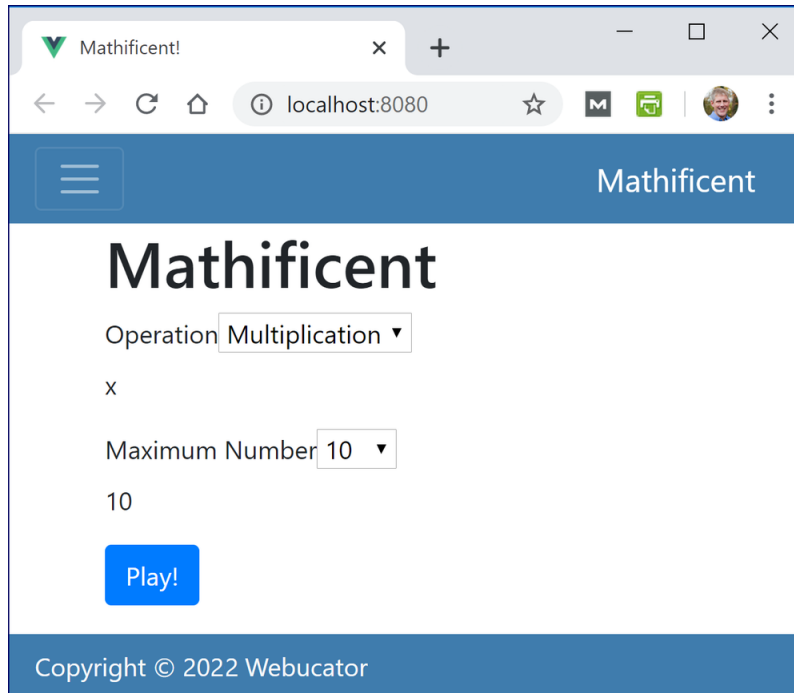
Evaluation
Copy

3. Add a `p` element below the `select` and output the value of `currentValue`:

```
<p>{{currentValue}}</p>
```

We are just putting this there temporarily to demonstrate how the current value changes.

4. At this point, each `SelectInput` element has its own internal state. If it is not already running, start up the Mathificent app and make selections from the two dropdowns. You should see something like this:



5. We need to be able to access the state of the `SelectInput` component from other components, so next we'll add an *event emitter* and cause the component to be controlled by its parent.
6. In `MainContainer.vue`, add `operation` and `maxNumber` to the `data` object and give them default values:

```
data: function() {  
  return {  
    operations: [  
      ['Addition', '+'],  
      ['Subtraction', '-'],  
      ['Multiplication', 'x'],  
      ['Division', '/']  
    ],  
    operation: 'x',  
    maxNumber: '10'  
  }  
},
```

7. Bind each instance of the `SelectInput` to a data property using `v-model` and pass that same property to `currentValue` as a prop:

```
<SelectInput :currentValue="operation" label="Operation"
  id="operation" v-model="operation" :options="operations" />
<SelectInput :currentValue="maxNumber" label="Maximum Number"
  id="max-number" v-model="maxNumber" :options="numbers" />
```

8. Inside the `MainContainer` template, below the `PlayButton`, output the values of the `operation` and `maxNumber` properties for testing:

```
<p>current operation: {{operation}}</p>
<p>max number: {{maxNumber}}</p>
```

9. Back in `SelectInput`, remove the data object with the `currentValue` property and add `currentValue` as a prop inside the `SelectInput` component.

```
export default {
  name: 'SelectInput',
  props: {
    id: String,
    label: String,
    options: Array,
    currentValue: String,
  }
}
```

Evaluation
Copy

Now, replace `v-model="currentValue"` in the select tag with `:value="currentValue"`, so that the dropdown gets its initial value from the parent:

```
<select :id="id" :value="currentValue">
```

10. Open your web browser and test out the app so far. **Some things to notice:**
- A. The `SelectInput` components have their own state and they receive initial values from the `MainContainer` component. You can see this by refreshing the page and noting that the **Operation** and **Maximum Number** dropdowns have pre-selected options.
 - B. However, changes in the `SelectInput` components aren't being reflected in `MainContainer`. You can see this by making a change to one of the dropdowns and noticing that the value under the **Play** button doesn't get updated. To enable

SelectInput to notify MainContainer of changes, we need to emit an event from SelectInput when an option is selected. And, then we need to listen for that event in MainContainer.

11. In the template for SelectInput, emit an event when a new option is selected and pass the value of the new selected option to MainContainer:

```
<select :id="id" :value="currentValue"
  @input="$emit('input', $event.target.value)">
```

\$event is the event that was fired. \$event.target is the element on which the event was fired: the select. And \$event.target.value is the current value of that element: the value of the option that was selected.

12. Style the select by applying a Bootstrap form-select class to it:

```
<select class="form-select" :id="id" :value="currentValue"
  @input="$emit('input', $event.target.value)">
```

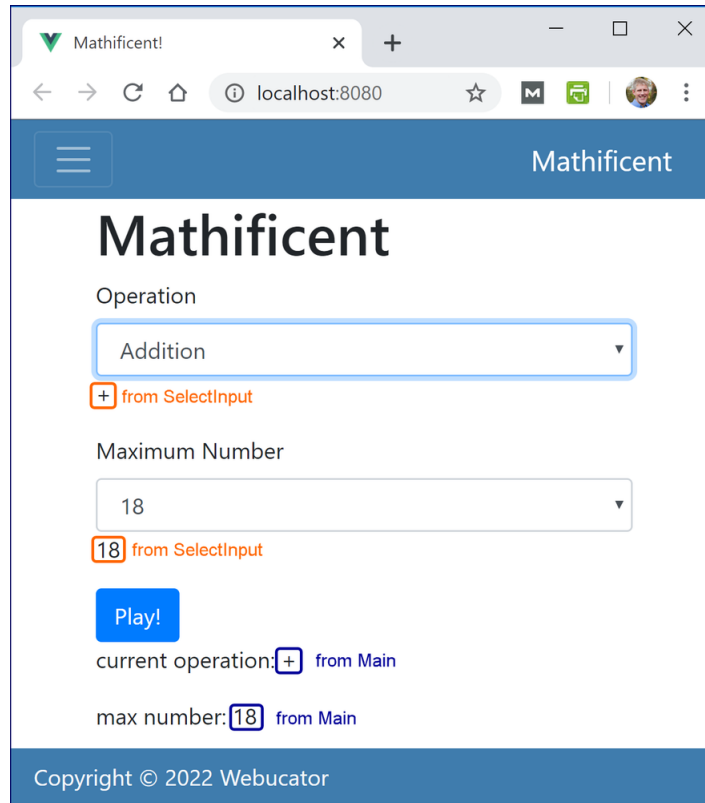
13. Add a emits property to the component's export and give it a value of an array with a single element containing the string input:

```
export default {
  name: 'SelectInput',
  emits: ['input'],
```

14. In the `MainContainer` component, add a `v-on` directive to each of the `SelectInput` elements to listen for the input event. The value of each directive will be a function that updates the relevant property in the `MainContainer`

```
<SelectInput
  :currentValue="operation"
  label="Operation"
  id="operation"
  v-model="operation"
  :options="operations"
  @input="(o) => (this.operation = o)"
/>
<SelectInput
  :currentValue="maxNumber"
  label="Maximum Number"
  id="max-number"
  v-model="maxNumber"
  :options="numbers"
  @input="(n) => (this.maxNumber = n)"
/>
```

15. Your application should now look like this (after changing the operation to addition and the maximum number to 18):



Change the values of the `select` menus and notice that both the values directly below them (from the `SelectInput` component) and the values below the **Play** button (from the `MainContainer` component) get updated to reflect the change.

Solution:

Vue/Solutions/implementing-game/data-binding/MainContainer.vue

```
1. <template>
2.   <main id="main-container">
3.     <h1>Mathificent</h1>
4.     <SelectInput :currentValue="operation" label="Operation"
5.       id="operation" v-model="operation" :options="operations" @in
        put="(o)=>(this.operation = o)" />
6.     <SelectInput :currentValue="maxNumber" label="Maximum Number"
7.       id="max-number" v-model="maxNumber" :options="numbers" @in
        put="(m)=>(this.maxNumber = m)" />
8.     <PlayButton />
9.     <p>current operation: {{operation}}</p>
10.    <p>max number: {{maxNumber}}</p>
11.  </main>
12. </template>
13.
14. <script>
15.   import SelectInput from './SelectInput';
16.   import PlayButton from './PlayButton';
17.
18.   export default {
19.     name: 'MainContainer',
20.     components: {
21.       SelectInput,
22.       PlayButton
23.     },
24.     data: function() {
25.       return {
26.         operations: [
27.           ['Addition', '+'],
28.           ['Subtraction', '-'],
29.           ['Multiplication', 'x'],
30.           ['Division', '/']
31.         ],
32.         operation: 'x',
33.         maxNumber: '10'
34.       }
35.     },
36.     computed: {
37.       numbers: function() {
38.         const numbers = [];
39.         for (let number = 2; number <= 100; number++) {
40.           numbers.push([number, number]);
41.         }

```

```
42.         return numbers;
43.     }
44. }
45. }
46. </script>
47.
48. <style scoped>
49.   #main-container {
50.     margin: auto;
51.     width: 380px;
52.   }
53. </style>
```

Solution: Vue/Solutions/implementing-game/data-binding/SelectInput.vue

```
1. <template>
2.   <div>
3.     <label :for="id">{{label}}</label>
4.     <select class="form-select" :id="id" :value="currentValue"
5.       @input="$emit('input', $event.target.value)">
6.       <option v-for="option in options" :key="option[1]"
7.         :value="option[1]">{{option[0]}}</option>
8.     </select>
9.     <p>{{currentValue}}</p>
10.   </div>
11. </template>
12.
13. <script>
14.   export default {
15.     name: 'SelectInput',
16.     emits: ['input'],
17.     props: {
18.       id: String,
19.       label: String,
20.       options: Array,
21.       currentValue: String
22.     }
23.   }
24. </script>
```

Exercise 8: Implementing Conditional Rendering

🕒 25 to 40 minutes

In this exercise, you will use conditional rendering to switch between the game configuration and the game play screens.

1. First, let's do a little cleanup:
 - A. In the `MainContainer` component, remove the two testing `<p>` tags below the `<PlayButton>` tag.
 - B. In the `SelectInput` component, remove the `<p>` tag below the `<select>` tag.
2. In the `MainContainer` component, add a new `screen` variable to the `data` object and set its initial value to `"config"`. Your `data` function should now look like this:

```
data: function() {  
  return {  
    operations: [  
      ['Addition', '+'],  
      ['Subtraction', '-'],  
      ['Multiplication', 'x'],  
      ['Division', '/']  
    ],  
    operation: 'x',  
    maxNumber: '10',  
    screen: 'config'  
  }  
}
```

3. Inside of the main element, add a div element with the id of “config-container” and a v-if directive to test whether the current value of screen is “config”:

```
<main id="main-container">
  <div v-if="screen === 'config'" id="config-container">
    <h1>Mathificent</h1>
    <SelectInput :currentValue="operation" label="Operation"
      id="operation" v-model="operation" :options="operations" @in
put="(o)=>(this.maxNumber = o)"/>
    <SelectInput :currentValue="maxNumber" label="Maximum Number"
      id="max-number" v-model="maxNumber" :options="numbers" @in
put="(m)=>(this.maxNumber = m)"/>
    <PlayButton />
  </div>
</main>
```

At this point, the value of the screen property will always be “config” so the code inside the div with the v-if directive will always be displayed. But we’re going to change that.

4. Below the div you just added, add another div with the id of “game-container” and a v-else-if directive. This div should display when the value of screen is “play”. Also, give this div the Bootstrap class of “text-center”. Inside the div, add the placeholder text “Game Here”:

```
<div v-else-if="screen === 'play'" id="game-container" class="text-center">
  Game Here
</div>
```

5. If it isn’t running already, start the development server and open `http://localhost:8080` in your web browser. The configuration screen should be displaying.
6. While the development server is still running, open `MainContainer.vue` in your editor and change the value of the screen property to “play”. The browser should now display the placeholder text for the game.
7. If everything works correctly, change the initial value of the screen property back to “config” so that we can code a more dynamic way of changing it.

8. Add a new property to the export statement, called `methods`. Methods are the functions that your application uses to change its data properties:

```
export default {  
  name: 'MainContainer',  
  components: {  
    ...  
  },  
  data: function() {  
    ...  
  },  
  methods: {  
  
  },  
  computed: {  
    ...  
  }  
}
```

9. Add two new functions inside the `methods` object: `config` and `play`. Inside these methods, change the value of `this.screen` to the correct values to change what is displayed.

```
methods: {  
  config() {  
    this.screen = "config";  
  },  
  play() {  
    this.screen = "play";  
  }  
},
```

10. Add a `v-on` directive to the button element in `PlayButton` that emits a custom event:

```
<button class="btn btn-primary" @click="$emit('play-button-click')">  
  Play!  
</button>
```

11. Back in `MainContainer`, call `play` when a `play-button-click` event occurs:

```
<PlayButton @play-button-click="play" />
```

12. Still in `MainContainer.vue`, replace the “Game Here” text with a new “Change Game” button directly inside the conditional `div` for the “play” screen, and add a `v-on` directive to it to call the `config()` method when clicked:

```
<div v-else-if="screen === 'play'" id="game-container" class="text-center">  
  <button class="btn btn-success" @click="config">Change Game</button>  
</div>
```

13. In your browser, click the “Play” button and the “Change Game” button to switch between the **Play** screen and the **Config** screen.

Solution:

Vue/Solutions/implementing-game/conditional-rendering/MainContainer.vue

```
1. <template>
2.   <main id="main-container">
3.     <div v-if="screen === 'config'" id="config-container">
4.       <h1>Mathificent</h1>
5.       <SelectInput :currentValue="operation" label="Operation"
6.         id="operation" v-model="operation" :options="operations" @in
          put="(o)=>(this.operation = o)"/>
7.       <SelectInput :currentValue="maxNumber" label="Maximum Number"
8.         id="max-number" v-model="maxNumber" :options="numbers" @in
          put="(m)=>(this.maxNumber = m)"/>
9.       <PlayButton @play-button-click="play" />
10.    </div>
11.    <div v-else-if="screen === 'play'" id="game-container" class="text-center">
12.      <button class="btn btn-success" @click="config">Change Game</button>
13.    </div>
14.  </main>
15. </template>
16.
17. <script>
18.   import SelectInput from './SelectInput';
19.   import PlayButton from './PlayButton';
20.
21.   export default {
22.     name: 'MainContainer',
23.     -----Lines 23 through 26 Omitted-----
27.     data: function() {
28.       return {
29.         operations: [
30.           ['Addition', '+'],
31.           ['Subtraction', '-'],
32.           ['Multiplication', 'x'],
33.           ['Division', '/']
34.         ],
35.         operation: 'x',
36.         maxNumber: '10',
37.         screen: 'config'
38.       }
39.     },
40.     methods: {
41.       config() {
42.         this.screen = "config";
43.       },
44.       play() {
```




```
45.         this.screen = "play";
46.     }
47. },
-----Lines 48 through 65 Omitted-----
```

Solution:

Vue/Solutions/implementing-game/conditional-rendering/PlayButton.vue

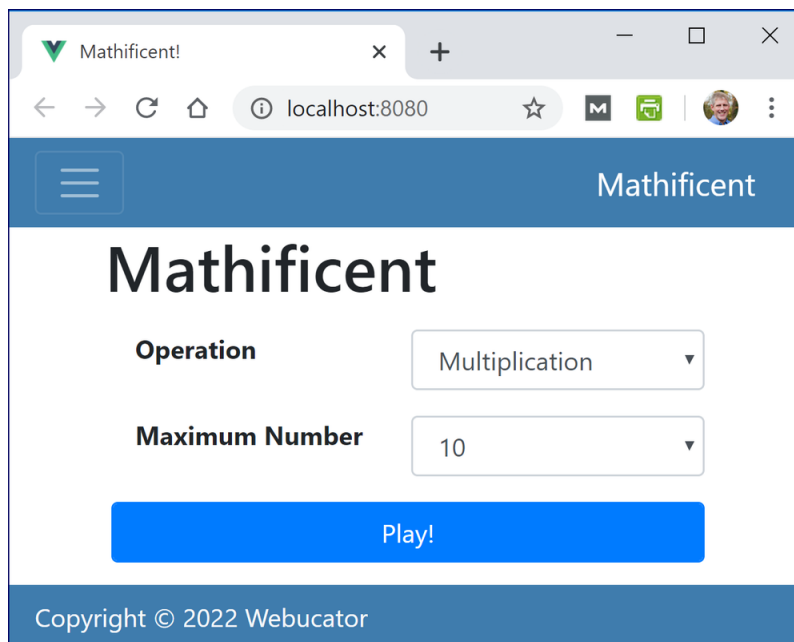
```
1. <template>
2.   <button class="btn btn-primary" @click="$emit('play-button-click')">
3.     Play!
4.   </button>
5. </template>
6.
7. <script>
8.   export default {
9.     name: 'PlayButton'
10.  }
11. </script>
```

Exercise 9: Improving the Form Layout

 10 to 15 minutes

In this exercise, we'll quickly improve the form layout by breaking the components into Bootstrap rows and columns. Keep Mathificent running on the **Config** screen, so you can watch it update as you make these changes.

1. In the template of the `SelectInput` component:
 - A. Add the `row`, `mx-1`, and `my-3` classes to the outer `div`. The `row` class makes the browser treat the `div` as a row. The `mx-1` and `my-3` classes give the row horizontal and vertical margins, respectively.
 - B. Add the `col` and `fw-bold` classes to the `label` element.
 - C. Add the `col` class to the `select` element.
2. In the template of the `PlayButton` component:
 - A. Wrap the button in a `div` element and give the `div` the `row`, `mx-1`, and `my-3` classes.
 - B. Add the `form-control` class to the button element.
3. The page should now look like this:



EVALUATION COPY: Not to be used in class.

Solution: Vue/Solutions/implementing-game/form-layout/SelectInput.vue

```
1. <template>
2.   <div class="row mx-1 my-3">
3.     <label :for="id" class="col fw-bold">{{label}}</label>
4.     <select class="col form-select" :id="id" :value="currentValue"
5.       @input="$emit('input', $event.target.value)">
6.       <option v-for="option in options" :key="option[1]"
7.         :value="option[1]">
8.         {{option[0]}}
9.       </option>
10.    </select>
11.  </div>
12. </template>
13.
14. <script>
15.   export default {
16.     name: 'SelectInput',
17.     emits: ['input'],
18.     props: {
19.       id: String,
20.       label: String,
21.       options: Array,
22.       currentValue: String
23.     }
24.   }
25. </script>
```



Solution: Vue/Solutions/implementing-game/form-layout/PlayButton.vue

```
1. <template>
2.   <div class="row mx-1 my-3">
3.     <button class="form-control btn btn-primary"
4.       @click="$emit('play-button-click')">
5.       Play!
6.     </button>
7.   </div>
8. </template>
9.
10. <script>
11.   export default {
12.     name: 'PlayButton'
13.   }
14. </script>
```

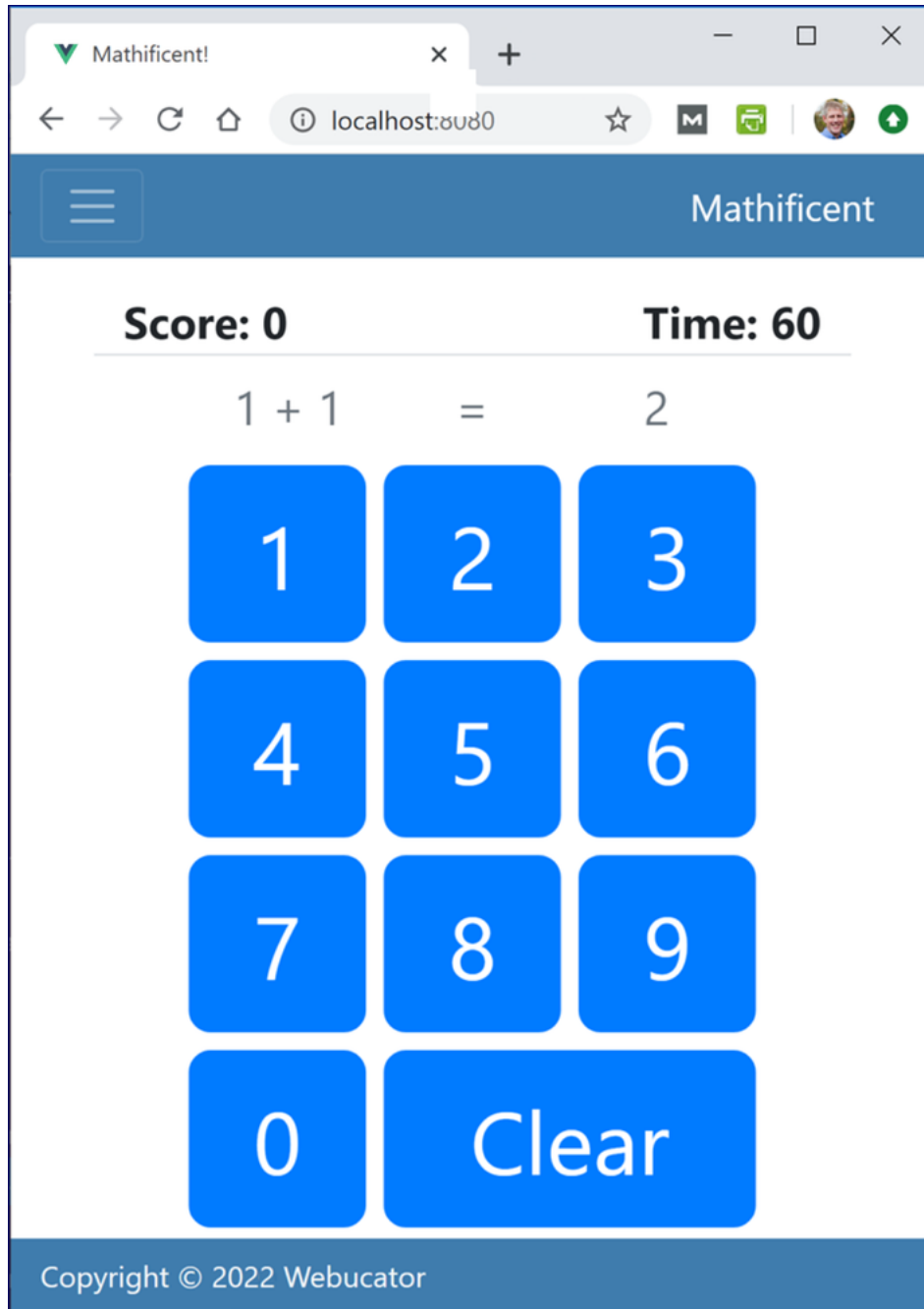


Exercise 10: Making the Game UI

⌚ 60 to 90 minutes

In this exercise, you will start building the user interface for the game. To see how the final game works, stop the development server if you have it running. Then run `npm install` followed by `npm run serve` from the `mathificent-final` directory and play around. Be sure to stop the server before moving on to the exercise.

1. Here is what the game will look like when it is complete:

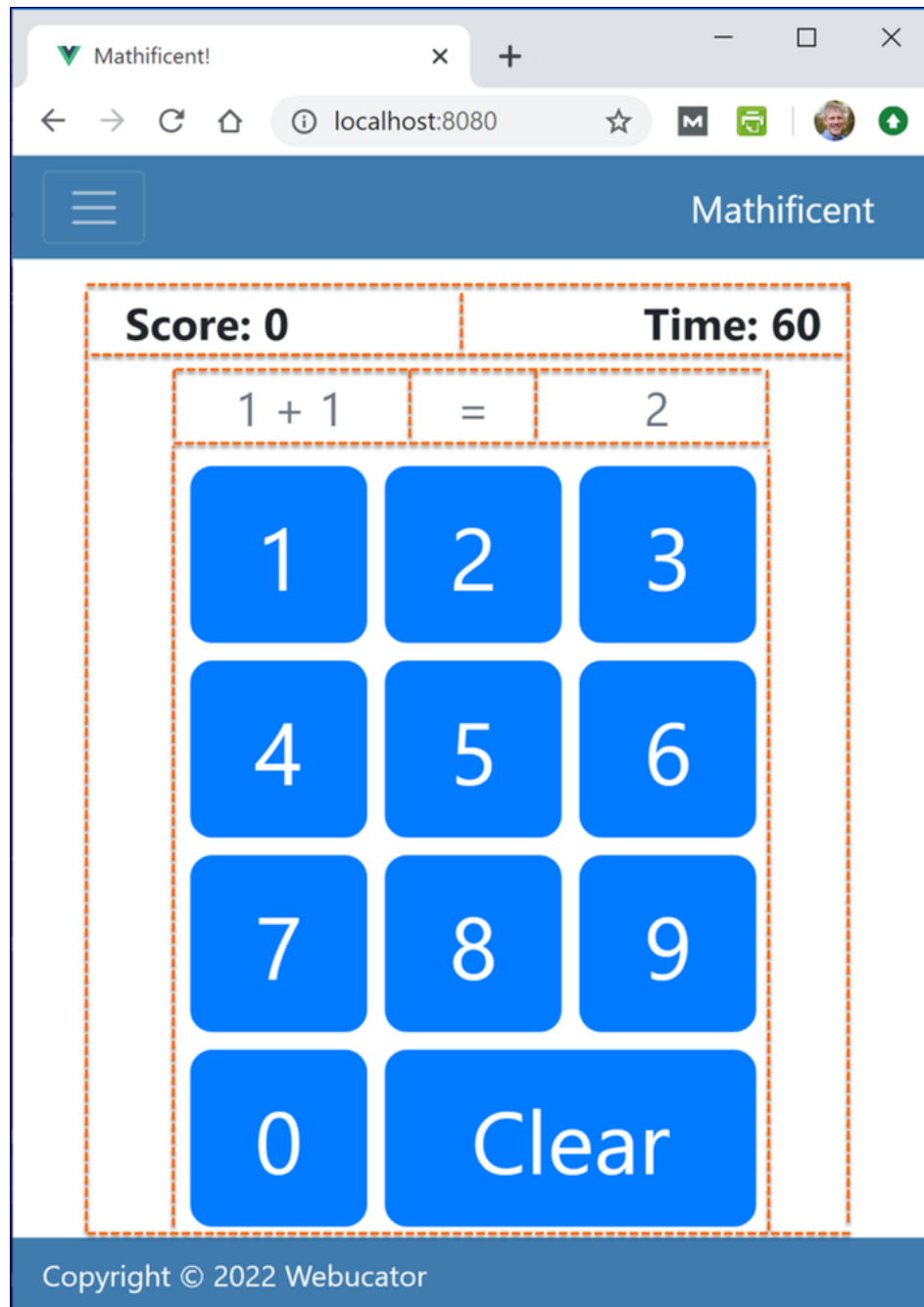


Using this screenshot, try to identify the individual elements that make up the user interface of the game.

2. Think about how many different components you need to make to build this user interface. It has:
 - A. A score.

- B. A timer.
- C. An equation.
- D. Ten number buttons.
- E. A clear button.

You could lay that out in rows and columns like this:



Note that the buttons are all in a single row, but are constrained by the width of the container.

3. Start up your development server in Exercises/mathifificent if it is not running already. Press the **Play** button to show the game. **As you proceed through the exercise, keep an eye on the web browser to see how the interface changes.**
4. Create placeholder components in the components directory for each of the following components (sample code is shown for the GameScore component):

A. GameScore

```
<template>
  <div>GameScore Component</div>
</template>

<script>
  export default {
    name: 'GameScore'
  }
</script>
```

B. GameTimer

C. GameEquation

For the number and clear buttons, we'll just use standard HTML button elements.

5. In the `MainContainer` component, remove the **Change Game** button, import the components you just created, and display them in Bootstrap rows in the template using the tags and classes shown in the following code:

```
...
<div v-else-if="screen === 'play'" id="game-container" class="text-center">
  <div class="row border-bottom" id="scoreboard">
    <div class="col px-3 text-left">
      <GameScore />
    </div>
    <div class="col px-3 text-right">
      <GameTimer />
    </div>
  </div>
  <div class="row text-secondary my-2" id="equation">
    <GameEquation />
  </div>
  <div class="row" id="buttons">
    <div class="col">
      <button class="btn btn-primary number-button">1</button>
      <button class="btn btn-primary number-button">2</button>
      ...
      <button class="btn btn-primary number-button">9</button>
      <button class="btn btn-primary number-button">0</button>
      <button class="btn btn-primary" id="clear-button">Clear</button>
    </div>
  </div>
</div>
```

```

...
import SelectInput from './SelectInput';
import PlayButton from './PlayButton';
import GameScore from './GameScore';
import GameTimer from './GameTimer';
import GameEquation from './GameEquation';

export default {
  name: 'MainContainer',
  components: {
    SelectInput,
    PlayButton,
    GameScore,
    GameTimer,
    GameEquation
  },
  ...

```

6. Write and style the game components' subcomponents using HTML, CSS, and Bootstrap classes:

Exercise Code 10.1: GameScore.vue

```

1. <template>
2.   <strong>Score: 0</strong>
3. </template>
4.
5. <script>
6.   export default {
7.     name: 'GameScore'
8.   }
9. </script>

```

Exercise Code 10.2: GameTimer.vue

```

1. <template>
2.   <strong>Time Left: 60</strong>
3. </template>
4.
5. <script>
6.   export default {
7.     name: 'GameTimer'
8.   }
9. </script>

```

Exercise Code 10.3: GameEquation.vue

```
1. <template>
2.   <div id="equation" class="row">
3.     <div class="col-5">1+1</div>
4.     <div class="col-2">=</div>
5.     <div class="col-5">2</div>
6.   </div>
7. </template>
8.
9. <script>
10.   export default {
11.     name: 'GameEquation'
12.   }
13. </script>
14.
15. <style scoped>
16.   #equation {
17.     font-size: 1.6em;
18.     margin: auto;
19.     width: 90%;
20.   }
21. </style>
```

Evaluation
Copy

Notice that the style tag includes a scoped attribute. This is so that the style rules only affect the component in which they are defined.

- Next, instead of hardcoding ten number buttons, we'll dynamically generate the tags using the v-for directive. Make a new property in the data object in `MainContainer.vue` called `buttons` and set its value to the numbers 1-9 and then a 0:

```
buttons: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

- Add a v-for directive to the button element to iterate over the `buttons` array:

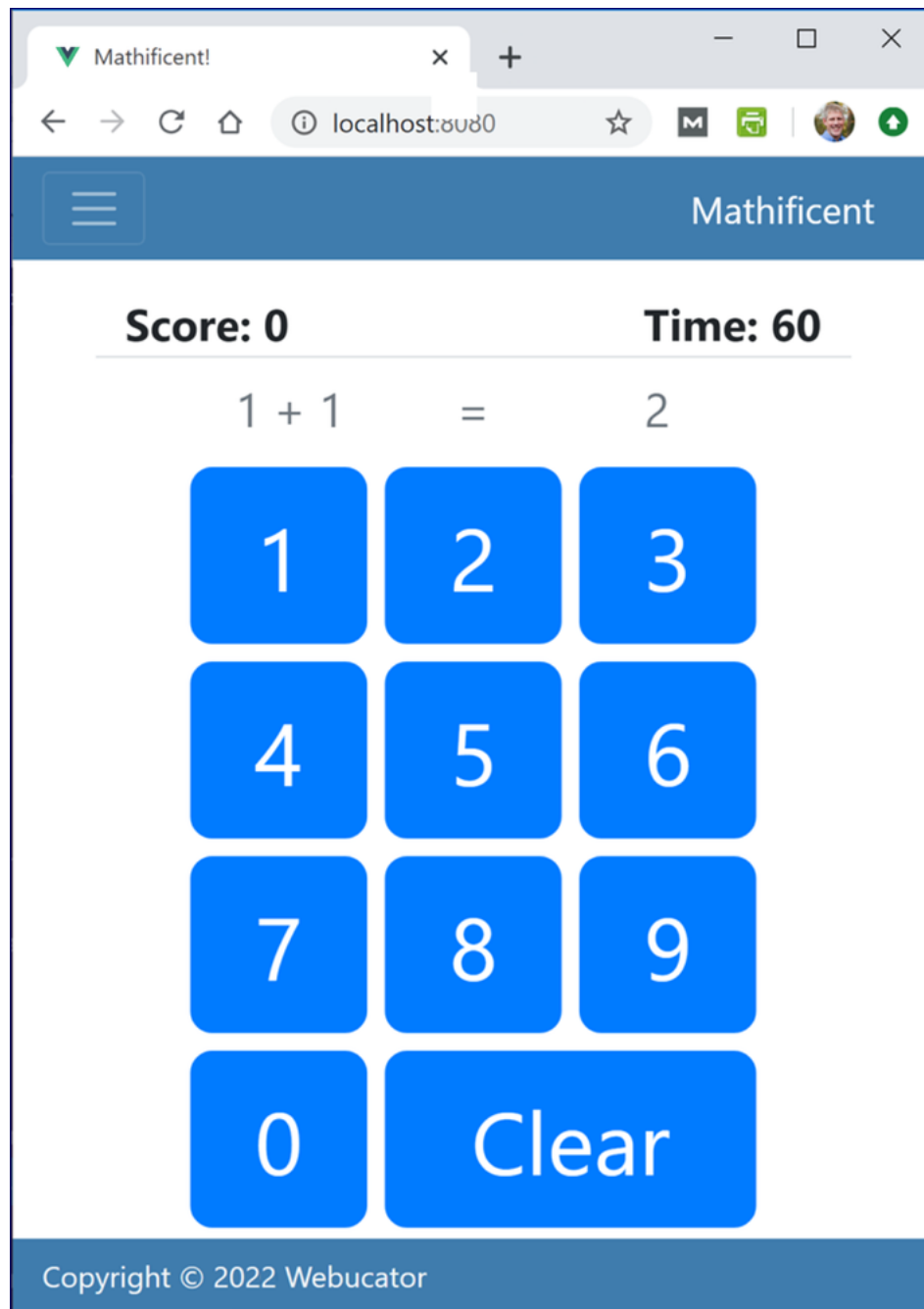
```
<button class="btn btn-primary number-button"
  v-for="button in buttons" :key="button">{{button}}</button>
```

9. In the MainContainer component's style block, add these rules to style the buttons and scoreboard:

```
button.number-button {  
  border-radius: .25em;  
  font-size: 3em;  
  height: 2em;  
  margin: .1em;  
  text-align: center;  
  width: 2em;  
}  
  
#clear-button {  
  border-radius: .25em;  
  font-size: 3em;  
  height: 2em;  
  margin: .1em;  
  text-align: center;  
  width: 4.2em;  
}  
  
#scoreboard {  
  font-size: 1.5em;  
}
```

Evaluation
Copy

10. The page should now look like this:



Solution: Vue/Solutions/implementing-game/game-ui/MainContainer.vue

```
1.   <template>
2.     <main id="main-container">
3.       <div v-if="screen === 'config'" id="config-container">
4.         <h1>Mathificent</h1>
5.         <SelectInput :currentValue="operation" label="Operation"
6.           id="operation" v-model="operation" :options="operations" @in
           put="(o)=>(this.operation = o)" />
7.         <SelectInput :currentValue="maxNumber" label="Maximum Number"
8.           id="max-number" v-model="maxNumber" :options="numbers" @in
           put="(m)=>(this.maxNumber = m)" />
9.         <PlayButton @play-button-click="play" />
10.      </div>
11.      <div v-else-if="screen === 'play'" id="game-container" class="text-center">
12.        <div class="row border-bottom" id="scoreboard">
13.          <div class="col px-3 text-left">
14.            <GameScore />
15.          </div>
16.          <div class="col px-3 text-right">
17.            <GameTimer />
18.          </div>
19.        </div>
20.        <div class="row text-secondary my-2" id="equation">
21.          <GameEquation />
22.        </div>
23.        <div class="row" id="buttons">
24.          <div class="col">
25.            <button class="btn btn-primary number-button"
26.              v-for="button in buttons" :key="button">{{button}}</button>
27.            <button class="btn btn-primary">Clear</button>
28.          </div>
29.        </div>
30.      </div>
31.    </main>
32.  </template>
33.
34.  <script>
35.    import SelectInput from './SelectInput';
36.    import PlayButton from './PlayButton';
37.    import GameScore from './GameScore';
38.    import GameTimer from './GameTimer';
39.    import GameEquation from './GameEquation';
40.
41.    export default {
42.      name: 'MainContainer',
```

```

43.     components: {
44.         SelectInput,
45.         PlayButton,
46.         GameScore,
47.         GameTimer,
48.         GameEquation
49.     },
50.     data: function() {
51.         return {
52.             operations: [
53.                 ['Addition', '+'],
54.                 ['Subtraction', '-'],
55.                 ['Multiplication', 'x'],
56.                 ['Division', '/']
57.             ],
58.             operation: 'x',
59.             maxNumber: '10',
60.             buttons: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
61.             screen: 'config'
62.         }
63.     },
64.     methods: {
65.         config() {
66.             this.screen = "config";
67.         },
68.         play() {
69.             this.screen = "play";
70.         }
71.     },
72.     computed: {
73.         numbers: function() {
74.             const numbers = [];
75.             for (let number = 2; number <= 100; number++) {
76.                 numbers.push([number, number]);
77.             }
78.             return numbers;
79.         }
80.     }
81. }
82. </script>
83.
84. <style scoped>
85.     #main-container {
86.         margin: auto;
87.         width: 380px;

```

```
88.     }
89.
90.     button.number-button {
91.         border-radius: .25em;
92.         font-size: 3em;
93.         height: 2em;
94.         margin: .1em;
95.         text-align: center;
96.         width: 2em;
97.     }
98.
99.     #clear-button {
100.         border-radius: .25em;
101.         font-size: 3em;
102.         height: 2em;
103.         margin: .1em;
104.         text-align: center;
105.         width: 4.2em;
106.     }
107.
108.     #scoreboard {
109.         font-size: 1.5em;
110.     }
111. </style>
```

Evaluation
Copy



Exercise 11: Capturing Form Events

⌚ 20 to 30 minutes

In this exercise, you will control the user input by capturing button clicks.

1. Add an input property to the data object in `MainContainer` and give it a default value of an empty string (`' '`). This will hold the user input generated through button clicks.
2. Add a `setInput()` method to the `MainContainer` component's methods that appends a passed-in string (value) to the input property:

```
setInput(value) {  
  this.input += String(value);  
  this.input = String(Number(this.input));  
}
```

`this.input` holds a string, so we first convert the value passed to `setInput()` to a string and append it to `this.input`. For example, if `this.input` is `'5'` and 2 is passed to `setInput()`, we convert 2 to `'2'` by passing it to `String()` and then we append `'2'` to `'5'` to give us `'52'`. This works great most of the time. The one exception is when `this.input` already holds `'0'`. Then if you pass 2, you will wind up with `'02'`. For that reason, we do this:

```
String(Number(this.input))
```

`Number('02')` will convert `'02'` to 2, and `String(2)` will convert 2 to `'2'`, which looks better than `'02'` in our game.

3. Make each number button element call `setInput()` and pass its value when clicked:

```
<button class="btn btn-primary number-button"  
  v-for="button in buttons" :key="button"  
  @click="setInput(button)">{{button}}</button>
```

4. Pass the value of input into the `GameEquation` component as a prop named `answer`, and modify the `GameEquation` component to display it in the place where the answer should be displayed. No code is shown for this one. See if you can do it on your own.

5. Add a method named `clear` to the `MainContainer` component:

```
clear() {  
  this.input = '';  
}
```

6. Make the clear button call `clear()` when clicked:

```
<button class="btn btn-primary" id="clear-button"  
  @click="clear">Clear</button>
```

7. In the app in the browser, click the **Play** button, and test whether clicking the number buttons adds the number to the displayed answer and whether clicking the **Clear** button resets the answer value.

Solution:

Vue/Solutions/implementing-game/capturing-events/MainContainer.vue

```
1. <template>
2.   <main id="main-container">
3.     <div v-if="screen === 'config'" id="config-container">
4.       <h1>Mathifient</h1>
5.       <SelectInput :currentValue="operation" label="Operation"
6.         id="operation" v-model="operation" :options="operations" @in
          put="(o)=>(this.operation = o)"/>
7.       <SelectInput :currentValue="maxNumber" label="Maximum Number"
8.         id="max-number" v-model="maxNumber" :options="numbers" @in
          put="(m)=>(this.maxNumber = m)"/>
9.       <PlayButton @play-button-click="play" />
10.    </div>
11.    <div v-else-if="screen === 'play'" id="game-container" class="text-center">
12.      <div class="row border-bottom" id="scoreboard">
13.        <div class="col px-3 text-left">
14.          <GameScore />
15.        </div>
16.        <div class="col px-3 text-right">
17.          <GameTimer />
18.        </div>
19.      </div>
20.      <div class="row text-secondary my-2" id="equation">
21.        <GameEquation :answer="input" />
22.      </div>
23.      <div class="row" id="buttons">
24.        <div class="col">
25.          <button class="btn btn-primary number-button"
26.            v-for="button in buttons" :key="button"
27.            @click="setInput(button)">{{button}}</button>
28.          <button class="btn btn-primary" id="clear-button"
29.            @click="clear">Clear</button>
30.        </div>
31.      </div>
32.    </div>
33.  </main>
34. </template>
35.
36. <script>
37.   import SelectInput from './SelectInput';
38.   import PlayButton from './PlayButton';
39.   import GameScore from './GameScore';
40.   import GameTimer from './GameTimer';
41.   import GameEquation from './GameEquation';
```

```

42.
43. export default {
44.   name: 'MainContainer',
45.   components: {
46.     SelectInput,
47.     PlayButton,
48.     GameScore,
49.     GameTimer,
50.     GameEquation
51.   },
52.   data: function() {
53.     return {
54.       operations: [
55.         ['Addition', '+'],
56.         ['Subtraction', '-'],
57.         ['Multiplication', 'x'],
58.         ['Division', '/']
59.       ],
60.       operation: 'x',
61.       maxNumber: '10',
62.       screen: 'config',
63.       buttons: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
64.       input: ''
65.     }
66.   },
67.   methods: {
68.     config() {
69.       this.screen = "config";
70.     },
71.     play() {
72.       this.screen = "play";
73.     },
74.     setInput(value) {
75.       this.input += String(value);
76.       this.input = String(Number(this.input));
77.     },
78.     clear() {
79.       this.input = '';
80.     }
81.   },

```

-----Lines 82 through 121 Omitted-----

Solution:

Vue/Solutions/implementing-game/capturing-events/GameEquation.vue

```
1.   <template>
2.     <div id="equation" class="row">
3.       <div class="col-5">1+1</div>
4.       <div class="col-2">=</div>
5.       <div class="col-5">{{answer}}</div>
6.     </div>
7.   </template>
8.
9.   <script>
10.    export default {
11.      name: 'GameEquation',
12.      props: {
13.        answer: String
14.      }
15.    }
16.  </script>
    -----Lines 17 through 24 Omitted-----
```

Evaluation
Copy



Exercise 12: Setting the Equation

⌚ 30 to 45 minutes

In this exercise, you will write the code to create the equations displayed in Mathificent.

1. We will need to generate random integers for the equation. The function for generating random integers is not specific to Mathificent, so we will put it in a separate `helpers.js` file and import it:
 - A. Create a new folder within the `src` folder called `helpers`.
 - B. Within the `helpers` folder, create a file called `helpers.js`.
 - C. In the `Exercises/starter-code.txt` file, you will find a JavaScript function called `randInt()` that looks like this:

```
export function randInt(low, high) {  
  const rndDec = Math.random();  
  return Math.floor(rndDec * (high - low + 1) + low);  
}
```

Copy and paste that code into `helpers.js` and save.

2. Open `MainContainer.vue` in your editor.
3. Import the `randInt()` function:

```
import {randInt} from '../helpers/helpers';
```

4. Add the following `getRandNumbers()` function to the `methods` property. This function is available to copy from `Exercises/starter-code.txt`, but be sure to review it so you understand how it works:

```
getRandNumbers(operator, low, high) {
  let num1 = randInt(low, high);
  let num2 = randInt(low, high);
  const numHigh = Math.max(num1, num2);
  const numLow = Math.min(num1, num2);

  if(operator === '-') { // Make sure higher num comes first
    num1 = numHigh;
    num2 = numLow;
  }

  if(operator === '/') {
    if (num2 === 0) { // No division by zero
      num2 = randInt(1, high);
    }
    num1 = (num1 * num2);
  }
  return {num1, num2};
}
```

5. Add two new properties, `operands` and `answered`, to the `data` object of `MainContainer`:
- `operands` will have a value that's an object containing two properties: the two operands¹ in the math problem.
 - `answered` indicates whether the user correctly answered the problem. It should default to `false`.

Here's the code to add:

```
operands: {num1: '1', num2: '1'},
answered: false
```

1. The *operands* are the numbers being operated on by the *operator*. In `5 + 3`, 5 and 3 are the operands.

6. Create a new method in the `MainContainer` component called `newQuestion`, which generates a new question:

```
newQuestion() {  
  this.input='';  
  this.answered = false;  
  this.operands = this.getRandNumbers(  
    this.operation, 0, this.maxNumber  
  );  
}
```

7. Add a new computed property that will use the operands and operation to generate an equation:

```
question: function() {  
  const num1 = this.operands.num1;  
  const num2 = this.operands.num2;  
  const equation = `${num1} ${this.operation} ${num2}`;  
  return equation;  
}
```

8. Pass `question` and `answered` into the `GameEquation` component as props and display the value of `question` in the correct place in `GameEquation`:

In `MainContainer.vue`

```
<GameEquation :question="question"  
  :answer="input"  
  :answered="answered" />
```

In `GameEquation.vue`

```
<div class="col-5">{{question}}</div>  
...  
props: {  
  question: String,  
  answer: String,  
  answered: Boolean  
}
```

9. Back in `MainContainer.vue`, add a call to `newQuestion` to the `play` method. This will cause a new question to be generated when the game starts:

```
play() {  
  this.screen = "play";  
  this.newQuestion();  
},
```

10. Add a check for the correct answer to the `setInput` method. If the answer is correct, get a new question. Your `setInput` method should now look like this:

```
setInput(value) {  
  this.input += String(value);  
  this.input = String(Number(this.input));  
  this.answered = this.checkAnswer(this.input,  
                                this.operation,  
                                this.operands);  
  
  if (this.answered) {  
    this.newQuestion();  
  }  
}
```

11. This will break the app, because we haven't added the `checkAnswer()` method yet. Copy the `checkAnswer()` method from the `starter-code.txt` file and add it to the

MainContainer component's methods. Review the function to make sure you understand it:

```
checkAnswer(userAnswer, operation, operands) {
  if (isNaN(userAnswer)) return false; // User hasn't answered

  let correctAnswer;
  switch(operation) {
    case '+':
      correctAnswer = operands.num1 + operands.num2;
      break;
    case '-':
      correctAnswer = operands.num1 - operands.num2;
      break;
    case 'x':
      correctAnswer = operands.num1 * operands.num2;
      break;
    default: // division
      correctAnswer = operands.num1 / operands.num2;
  }
  return (parseInt(userAnswer) === correctAnswer);
}
```

12. Make a new data property in MainContainer called score to keep track of the score and set its initial value to 0.
13. In setInput, increase the value of score when answered is true:

```
if (this.answered) {
  this.newQuestion();
  this.score++;
}
```

14. Pass score into the GameScore component, add it to the GameScore component's props, and display the score.
15. Try it out in the browser. You should now be able to answer question after question forever and ever.

16. The question changes are a little abrupt. It'd be nice to fade the old question out. We will use the Bootstrap "fade" class for this. In the `MainContainer` component, add a computed property called `equationClass` whose value depends on the value of `answered`:

```
equationClass: function() {  
  if (this.answered) {  
    return 'row text-primary my-2 fade';  
  } else {  
    return 'row text-secondary my-2';  
  }  
}
```

Now, change the class of the `div` containing the `<GameEquation>` tag to be bound to `equationClass`:

```
<div :class="equationClass" id="equation">  
  <GameEquation :question="question"  
    :answer="input"  
    :answered="answered" />  
</div>
```

Finally, we need to add a little delay before getting the next question so that the user has time to see the fade effect. In the `if` condition where you call `this.newQuestion()`, use a `setTimeout` to delay that call by 300 milliseconds:

```
if (this.answered) {  
  setTimeout(this.newQuestion, 300);  
  this.score++;  
}
```

Try out the app again. After answering a question correctly, the equation and answer should fade away before a new question shows up. Things are working pretty well! Time to implement our timer.

Remember to stop the app (**CTRL+C**) and close the terminal when you are done.

Solution:

Vue/Solutions/implementing-game/setting-the-equation/MainContainer.vue

```
1.   <template>
2.     <main id="main-container">
3.       -----Lines 3 through 10 Omitted-----
11.     <div v-else-if="screen === 'play'" id="game-container" class="text-center">
12.       <div class="row border-bottom" id="scoreboard">
13.         <div class="col px-3 text-left">
14.           <GameScore :score="score" />
15.         </div>
16.         <div class="col px-3 text-right">
17.           <GameTimer />
18.         </div>
19.       </div>
20.       <div :class="equationClass" id="equation">
21.         <GameEquation :question="question"
22.           :answer="input"
23.           :answered="answered" />
24.       </div>
25.       <div class="row" id="buttons">
26.         <div class="col">
27.           <button class="btn btn-primary number-button"
28.             v-for="button in buttons" :key="button"
29.             @click="setInput(button)">{{button}}</button>
30.           <button class="btn btn-primary" id="clear-button"
31.             @click="clear">Clear</button>
32.         </div>
33.       </div>
34.     </div>
35.   </main>
36. </template>
37.
38. <script>
39.   import SelectInput from './SelectInput';
40.   import PlayButton from './PlayButton';
41.   import GameScore from './GameScore';
42.   import GameTimer from './GameTimer';
43.   import GameEquation from './GameEquation';
44.   import {randInt} from '../helpers/helpers';
45.   export default {
46.     name: 'MainContainer',
47.     components: {
48.       SelectInput,
49.       PlayButton,
```

```

50.     GameScore,
51.     GameTimer,
52.     GameEquation
53. },
54. data: function() {
55.     return {
56.         operations: [
57.             ['Addition', '+'],
58.             ['Subtraction', '-'],
59.             ['Multiplication', 'x'],
60.             ['Division', '/']
61.         ],
62.         operation: 'x',
63.         maxNumber: '10',
64.         buttons: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
65.         screen: 'config',
66.         input: '',
67.         operands: {num1: '1', num2: '1'},
68.         answered: false,
69.         score: 0
70.     }
71. },
72. methods: {
73.     config() {
74.         this.screen = "config";
75.     },
76.     play() {
77.         this.screen = "play";
78.         this.newQuestion();
79.     },
80.     setInput(value) {
81.         this.input += String(value);
82.         this.input = String(Number(this.input));
83.         this.answered = this.checkAnswer(this.input,
84.                                           this.operation,
85.                                           this.operands);
86.         if (this.answered) {
87.             setTimeout(this.newQuestion, 300);
88.             this.score++;
89.         }
90.     },
91.     clear() {
92.         this.input = '';
93.     },
94.     getRandNumbers(operator, low, high) {

```

```

95.     let num1 = randInt(low, high);
96.     let num2 = randInt(low, high);
97.     const numHigh = Math.max(num1, num2);
98.     const numLow = Math.min(num1, num2);
99.
100.    if(operator === '-') { // Make sure higher num comes first
101.        num1 = numHigh;
102.        num2 = numLow;
103.    }
104.
105.    if(operator === '/') {
106.        if (num2 === 0) { // No division by zero
107.            num2 = randInt(1, high);
108.        }
109.        num1 = (num1 * num2);
110.    }
111.    return {num1, num2};
112. },
113. checkAnswer(userAnswer, operation, operands) {
114.     if (isNaN(userAnswer)) return false; // User hasn't answered
115.
116.     let correctAnswer;
117.     switch(operation) {
118.         case '+':
119.             correctAnswer = operands.num1 + operands.num2;
120.             break;
121.         case '-':
122.             correctAnswer = operands.num1 - operands.num2;
123.             break;
124.         case 'x':
125.             correctAnswer = operands.num1 * operands.num2;
126.             break;
127.         default: // division
128.             correctAnswer = operands.num1 / operands.num2;
129.     }
130.     return (parseInt(userAnswer) === correctAnswer);
131. },
132. newQuestion() {
133.     this.input = '';
134.     this.answered = false;
135.     this.operands = this.getRandNumbers(
136.         this.operation, 0, this.maxNumber
137.     );
138. }
139. },

```



```

140.     computed: {
141.       numbers: function() {
142.         const numbers = [];
143.         for (let number = 2; number <= 100; number++) {
144.           numbers.push([number, number]);
145.         }
146.         return numbers;
147.       },
148.       question: function() {
149.         const num1 = this.operands.num1;
150.         const num2 = this.operands.num2;
151.         const equation = `${num1} ${this.operation} ${num2}`;
152.         return equation;
153.       },
154.       equationClass: function() {
155.         if (this.answered) {
156.           return 'row text-primary my-2 fade';
157.         } else {
158.           return 'row text-secondary my-2';
159.         }
160.       }
161.     }
162.   }
163. </script>

```

-----Lines 164 through 191 Omitted-----

Solution:

Vue/Solutions/implementing-game/setting-the-equation/GameScore.vue

```


1.   <template>
2.     <strong>Score: {{score}}</strong>
3.   </template>
4.
5.   <script>
6.     export default {
7.       name: 'GameScore',
8.       props: {
9.         score: Number
10.      }
11.    }
12.   </script>

```

Solution:

Vue/Solutions/implementing-game/setting-the-equation/GameEquation.vue

```
1. <template>
2.   <div id="equation" class="row">
3.     <div class="col-5">{{question}}</div>
4.     <div class="col-2">=</div>
5.     <div class="col-5">{{answer}}</div>
6.   </div>
7. </template>
8.
9. <script>
10.  export default {
11.    name: 'GameEquation',
12.    props: {
13.      question: String,
14.      answer: String,
15.      answered: Boolean
16.    }
17.  }
18. </script>
-----Lines 19 through 26 Omitted-----
```



Solution:

Vue/Solutions/implementing-game/setting-the-equation/helpers/helpers.js

```
1. export function randInt(low, high) {
2.   const rndDec = Math.random();
3.   return Math.floor(rndDec * (high - low + 1) + low);
4. }
```

Conclusion

In this lesson, you have used your Vue and JavaScript skills to build out the Mathificent game. We have a couple of improvements to make, but the game is relatively usable at this point.

LESSON 5

Transitions and Animations

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ The transition Component.
- ☒ Adding a timer.

Introduction

Transitions and animations often make the difference between a user interface that's functional and a user interface that feels natural and looks great.

EVALUATION COPY: Not to be used in class.



5.1. Using the transition Component

Vue's transition component creates enter and leave transitions on elements as they appear in and are removed from view. In its simplest form, the transition component wraps around an element and takes a name attribute that will add and remove CSS classes at appropriate timings:

```
<transition name="fade">
  <h1 v-show="visible">Hello, world!</h1>
</transition>
```

The above code will cause a series of CSS classes to be added and removed:

1. fade-enter-from – this is the starting state for the entering transition. It's added one frame before the element is inserted and removed one frame after it's inserted.

2. `fade-enter-active` – this class is active during the entire entering transition.
3. `fade-enter-to` – this is the ending state for the transition. It's added at the same time as the `-enter` class is removed.
4. `fade-leave-from` – this is the the starting state for the exit transition.
5. `fade-leave-active` – this class is active during the entire exit transition.
6. `fade-leave-to` – this is the ending state of the exit transition.

❖ 5.1.1. Transitioning with CSS

By itself, the transition component doesn't add transitions, but by assigning CSS transitions or animations to these classes, you can create interesting effects. Here is a simple example:

Demo 5.1: Vue/demo-viewer/src/components/transitions/FadeComp.vue

```
1.  <template>
2.    <div class="container">
3.      <button class="btn btn-primary" @click="visible = !visible">
4.        {{toggleCommand}}
5.      </button>
6.      <transition name="fade">
7.        <h1 v-show="visible">Hello, world!</h1>
8.      </transition>
9.    </div>
10. </template>
11.
12. <script>
13.   export default {
14.     name: "FadeComp",
15.     data: function() {
16.       return {
17.         visible: false,
18.       }
19.     },
20.     computed: {
21.       toggleCommand: function() {
22.         return (this.visible ? 'Hide' : 'Show');
23.       }
24.     }
25.   }
26. </script>
27.
28. <style scoped>
29.   .fade-enter-from {
30.     opacity: 0;
31.   }
32.
33.   .fade-enter-active {
34.     transition: opacity 5s;
35.   }
36.
37.   .fade-enter-to {
38.     opacity: 1;
39.   }
40.
41.   .fade-leave-from {
42.     opacity: 1;
43.   }
44.
```

```
45.   .fade-leave-active {
46.     transition: opacity 5s;
47.   }
48.
49.   .fade-leave-to {
50.     opacity: 0;
51.   }
52. </style>
```

If the demo-viewer app isn't already running, run `npm run serve` from the `demo-viewer` directory and then open `http://localhost:8080` in your browser. Then click the **Fade Transition** link under **Transitions**. Click the **Show** button to see the “Hello, world!” text fade in. The button label will change to “Hide”. Click the **Hide** button to see the text fade out.

Fancy Transitions

Transitions can get really fancy. See <https://vuejs.org/guide/built-ins/transition.html> for some demos showing the cool things you can do by combining CSS and JavaScript with Vue's transition element.



Exercise 13: Adding the Timer

⌚ 15 to 25 minutes

We'll now finish the functionality of the game by adding a timer. When the timer runs out, the screen will switch from the game to a **Time's Up!** view. First, we'll make that switch abruptly, and then we will add a transition.

1. Add a new property in the data object of MainContainer named `gameLength` and give it a default value of 60.
2. Add a new data property named `timeLeft` and give it a default value of 0.
3. Add a new method named `startTimer()` that sets an interval that decrements `timeLeft` by 1 every second and then clears the timer when `timeLeft` is 0:

```
startTimer() {  
  this.timeLeft = this.gameLength;  
  if (this.timeLeft > 0) {  
    this.timer = setInterval(() => {  
      this.timeLeft--;  
      if (this.timeLeft === 0) {  
        clearInterval(this.timer);  
      }  
    }, 1000)  
  }  
}
```

4. Call `startTimer` from inside the `play` method.

```
play() {  
  this.screen = "play";  
  this.newQuestion();  
  this.startTimer();  
}
```

5. Add a new method named `restart()` that sets score to 0, restarts the timer, and gets a new question:

```
restart() {  
  this.score = 0;  
  this.startTimer();  
  this.newQuestion();  
}
```

Evaluation
Copy

6. Use the `v-if` and `v-else` directives to show a **Time's Up!** view if the value of `timeLeft` is 0. Note that you will be nesting template tags. The “game-container” div should now contain this code, some of which you can copy and paste from the `starter-code.txt` file:

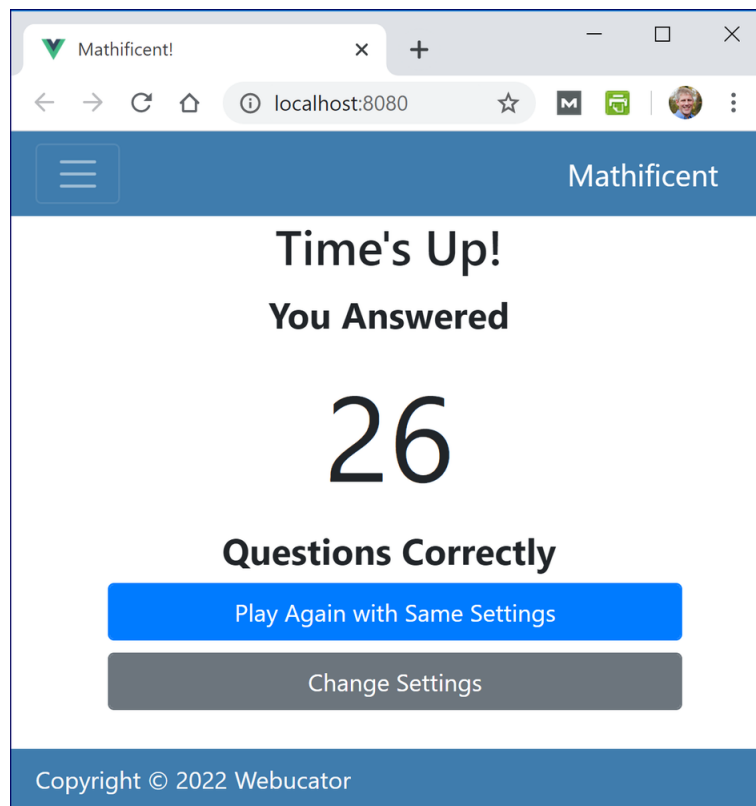
```
<template v-if="timeLeft === 0">
  <h2>Time's Up!</h2>
  <strong class="big">You Answered</strong>
  <div class="huge">{{score}}</div>
  <strong class="big">Questions Correctly</strong>
  <button class="btn btn-primary form-control m-1"
    @click="restart()">
    Play Again with Same Settings
  </button>
  <button class="btn btn-secondary form-control m-1"
    @click="config()">
    Change Settings
  </button>
</template>
<template v-else>
  <div class="row border-bottom" id="scoreboard">
    <div class="col px-3 text-left">
      <GameScore :score="score" />
    </div>
    <div class="col px-3 text-right">
      <GameTimer />
    </div>
  </div>
  <div :class="equationClass" id="equation">
    <GameEquation :question="question"
      :answer="input"
      :answered="answered" />
  </div>
  <div class="row" id="buttons">
    <div class="col">
      <button class="btn btn-primary number-button"
        v-for="button in buttons" :key="button"
        @click="setInput(button)">{{button}}</button>
      <button class="btn btn-primary" id="clear-button"
        @click="clear()">Clear</button>
    </div>
  </div>
</template>
```

7. Pass the value of `timeLeft` into the `GameTimer` component as a prop and display it.

8. Finally, add the following two classes, which are used in the **Times Up!** screen, to `MainContainer.vue`:

```
.big {  
  font-size: 1.5em;  
}  
  
.huge {  
  font-size: 5em;  
}
```

9. Try out the game. The timer should count down and when it reaches 0, the game should be replaced by the **Time's Up!** view:



Solution:

Vue/Solutions/transitions-animations/adding-timer/MainContainer.vue

```
1.   <template>
2.     <main id="main-container">
3.       -----Lines 3 through 10 Omitted-----
11.     <div v-else-if="screen === 'play'" id="game-container" class="text-center">
12.       <template v-if="timeLeft === 0">
13.         <h2>Time's Up!</h2>
14.         <strong class="big">You Answered</strong>
15.         <div class="huge">{{score}}</div>
16.         <strong class="big">Questions Correctly</strong>
17.         <button class="btn btn-primary form-control m-1"
18.           v-on:click="restart()">
19.           Play Again with Same Settings
20.         </button>
21.         <button class="btn btn-secondary form-control m-1"
22.           v-on:click="config()">
23.           Change Settings
24.         </button>
25.       </template>
26.       <template v-else>
27.         -----Lines 27 through 48 Omitted-----
49.       </template>
50.     </div>
51.   </main>
52. </template>
53.
54. <script>
55.   -----Lines 55 through 69 Omitted-----
70.   data: function() {
71.     return {
72.       -----Lines 72 through 84 Omitted-----
85.       score: 0,
86.       gameLength: 60,
87.       timeLeft: 0
88.     },
89.     },
90.     methods: {
91.       -----Lines 91 through 157 Omitted-----
158.       startTimer() {
159.         this.timeLeft = this.gameLength;
160.         if (this.timeLeft > 0) {
161.           this.timer = setInterval(() => {
```

Evaluation
Copy

```
162.         this.timeLeft--;
163.         if (this.timeLeft === 0) {
164.             clearInterval(this.timer);
165.         }
166.     }, 1000)
167. }
168. },
169. restart() {
170.     this.score = 0;
171.     this.startTimer();
172.     this.newQuestion();
173. }
174. },
```

-----Lines 175 through 195 Omitted-----

```
196.     }
197. }
198. </script>
199.
```

```
200. <style scoped>
```

-----Lines 201 through 227 Omitted-----

```
228.     .big {
229.         font-size: 1.5em;
230.     }
231.
232.     .huge {
233.         font-size: 5em;
234.     }
235. </style>
```

Solution:

Vue/Solutions/transitions-animations/adding-timer/GameTimer.vue

```
1. <template>
2.   <strong>Time Left: {{timeLeft}}</strong>
3. </template>
4.
5. <script>
6.   export default {
7.     name: 'GameTimer',
8.     props: {
9.       timeLeft: Number
10.    }
11.  }
12. </script>
```

Evaluation
Copy



Exercise 14: Adding Transitions

🕒 15 to 25 minutes

Now that the game is complete, we'll add a cool sliding animation to transition between the game screen and the "Time's Up" screen.

1. Put an opening `<transition>` tag just before the `<template>` tag that tests whether the time is up. Give this transition element a name attribute with a value of "slide".

```
<transition name="slide">
  <template v-if="timeLeft === 0">
```

2. Wrap the "Time's Up" screen in a `div` element. A transition can be applied to any element that's rendered conditionally (meaning inside a `v-if` or a `v-show`), but it's only applied to a single element. In the case of our "Time's Up" screen, we have several elements. By wrapping them all in a single `div` element, however, we create the single element that the transition will be applied to.

```
<div>
  <h2>Time's Up!</h2>
  ...
</button>
</div>
```

Evaluation
Copy

3. Close the transition element after the closing `template` tag.
4. Change the `template` opening tag following the "Time's Up" screen so that it uses a `v-if` instead of `v-else`. This is necessary because `v-else` only works if it's the next element after a closing tag for an element that has a `v-if` directive.

```
<template v-if="timeLeft > 0">
```

5. Add a transition element around the game screen (just before the `template` tag you just edited) and give it a name attribute with a value of "slide-right":

```
<transition name="slide-right">
  <template v-if="timeLeft > 0">
```

6. Wrap the code inside this `template` element with a `div` element. This is required, because the `transition` element expects exactly one child element.
7. Close the `transition` element after the closing `template` tag.
8. Start the development server and view the game as it is now. You'll notice that nothing has changed. To create the actual transition effects, we need to write some CSS.
9. Add the following CSS rules into the `style` section of the component. You can also copy them from the `starter-code.txt` file. These are the styles for the classes that will be added and removed during the transition of the "Time's Up" and game screens. Study these styles, along with the list of classes that get added and removed during a transition, and see if you can follow and predict what the transitions will look like.

```
.slide-leave-active,  
.slide-enter-active {  
  position: absolute;  
  top: 56px;  
  transition: 1s;  
  width: 380px;  
}  
.slide-enter-from {  
  transform: translate(-100%, 0);  
  transition: opacity .5s;  
}  
.slide-leave-to {  
  transform: translate(100%, 0);  
  opacity: 0;  
}  
.slide-right-leave-active,  
.slide-right-enter-active {  
  position: absolute;  
  top: 56px;  
  transition: 1s;  
  width: 380px;  
}  
.slide-right-enter-from {  
  transform: translate(100%, 0);  
  transition: opacity .5s;  
}  
.slide-right-leave-to {  
  transform: translate(-100%, 0);  
  opacity: 0;  
}
```

10. Start your development server, if isn't already running, and play the game! When your time is up, the game screen should slide and fade to the side, while the "Time's Up" screen slides and fades from the other side. When you start a new game from the "Time's Up" screen, the game screen will slide and fade into place. If you don't want to wait a full minute to see the transition, you can change the `gameLength` property to something short, like 5.

Solution:

Vue/Solutions/transitions-animations/adding-transitions/MainContainer.vue

```
1.   <template>
    -----Lines 2 through 10 Omitted-----
11.     <div v-else-if="screen === 'play'" id="game-container" class="text-center">
12.       <transition name="slide">
13.         <template v-if="timeLeft === 0">
14.           <div>
15.             <h2>Time's Up!</h2>
    -----Lines 16 through 27 Omitted-----
28.         </template>
29.       </transition>
30.       <transition name="slide-right">
31.         <template v-if="timeLeft > 0">
32.           <div>
33.             <div class="row border-bottom" id="scoreboard">
    -----Lines 34 through 40 Omitted-----
41.             <div :class="equationClass" id="equation">
    -----Lines 42 through 45 Omitted-----
46.             <div class="row" id="buttons">
    -----Lines 47 through 54 Omitted-----
55.           </div>
56.         </template>
57.       </transition>
58.     </div>
59.   </main>
60. </template>
    -----Lines 61 through 207 Omitted-----
208. <style scoped>
    -----Lines 209 through 242 Omitted-----
243.
244.   .slide-leave-active,
245.   .slide-enter-active {
246.     position: absolute;
247.     top: 56px;
248.     transition: 1s;
249.     width: 380px;
250.   }
251.
252.   .slide-enter-from {
253.     transform: translate(-100%, 0);
254.     transition: opacity .5s;
255.   }
```

Evaluation
Copy

```
256.
257.   .slide-leave-to {
258.     opacity:0;
259.     transform: translate(100%, 0);
260.   }
261.
262.   .slide-right-leave-active,
263.   .slide-right-enter-active {
264.     position: absolute;
265.     top: 56px;
266.     transition: 1s;
267.     width: 380px;
268.   }
269.
270.   .slide-right-enter-from {
271.     transform: translate(100%, 0);
272.     transition: opacity .5s;
273.   }
274.
275.   .slide-right-leave-to {
276.     opacity:0;
277.     transform: translate(-100%, 0);
278.   }
279. </style>
```



Exercise 15: Catching Keyboard Events

🕒 10 to 15 minutes

Finally, just to make our game a little more user friendly, let's catch keyboard events so the user can enter numbers with the keyboard.

1. In `MainContainer.vue`, add a new method called `handleKeyUp` that handles keyup events:

```
handleKeyUp(e) {  
  e.preventDefault(); // prevent the normal behavior of the key  
  if (e.keyCode === 32 || e.keyCode === 13) { // space/Enter  
    this.clear();  
  } else if (e.keyCode === 8) { // backspace  
    this.input = this.input.substring(0, this.input.length - 1);  
  } else if (!isNaN(e.key)) {  
    this.setInput(e.key);  
  }  
}
```

2. Now, add an event listener when the timer starts to capture keyup events, and remove this event listener when the timer clears:

```
startTimer() {  
  window.addEventListener('keyup', this.handleKeyUp);  
  this.timeLeft = this.gameLength;  
  if (this.timeLeft > 0) {  
    this.timer = setInterval(() => {  
      this.timeLeft--;  
      if (this.timeLeft === 0) {  
        clearInterval(this.timer);  
        window.removeEventListener('keyup', this.handleKeyUp);  
      }  
    }, 1000)  
  }  
}
```

3. Start your development server, if isn't already running, and play the game. You should be able to use your keyboard to answer questions. The **spacebar** and **Enter** keys should work like the **Clear** button, and the **Backspace** key should work to delete the last character added.

Solution:

Vue/Solutions/transitions-animations/catching-keyboard/MainContainer.vue

```
-----Lines 1 through 165 Omitted-----
166.     startTimer() {
167.         window.addEventListener('keyup', this.handleKeyUp);
168.         this.timeLeft = this.gameLength;
169.         if (this.timeLeft > 0) {
170.             this.timer = setInterval(() => {
171.                 this.timeLeft--;
172.                 if (this.timeLeft === 0) {
173.                     clearInterval(this.timer);
174.                     window.removeEventListener('keyup', this.handleKeyUp);
175.                 }
176.             }, 1000)
177.         }
178.     },
179.     restart() {
180.         this.score = 0;
181.         this.startTimer();
182.         this.newQuestion();
183.     },
184.     handleKeyUp(e) {
185.         e.preventDefault(); // prevent the normal behavior of the key
186.         if (e.keyCode === 32 || e.keyCode === 13) { // space/Enter
187.             this.clear();
188.         } else if (e.keyCode === 8) { // backspace
189.             this.input = this.input.substring(0, this.input.length - 1);
190.         } else if (!isNaN(e.key)) {
191.             this.setInput(e.key);
192.         }
193.     }
194. },
-----Lines 195 through 291 Omitted-----
```

Conclusion

In this lesson, you have learned how to use conditional rendering and transitions together to create dynamic effects and transitions in Vue user interfaces. You have also finished the Mathificent game by adding a timer and keyboard interaction. Congratulations!

LESSON 6

Vue 3 Routing

EVALUATION COPY: Not to be used in class.

Topics Covered

- ☒ Routing in a Vue 3 application.

Introduction

All the functionality of our Mathificent app is finished now. But, you may have noticed that the `MainContainer` component is rather large and complicated. One of the most important principles of designing a component-based application is that a component should only do one thing. `MainContainer` currently handles both the configuration and game for our app. We can start to make our application easier to understand and maintain by breaking apart the `MainContainer` component. We'll do this by creating separate components and by implementing routing. Routing refers to the ability to change what displays in the browser based on the current value of the browser `location` property. Another benefit of using routing in Mathificent is that it will give the user a way to return to the configuration screen at any time, rather than having to wait until the game timer reaches zero.

EVALUATION COPY: Not to be used in class.

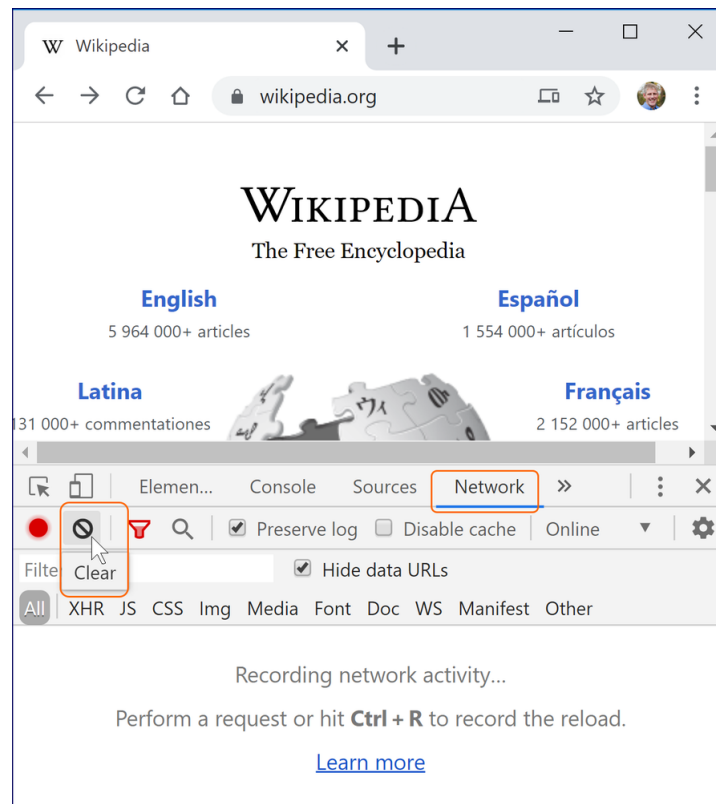


6.1. Routing

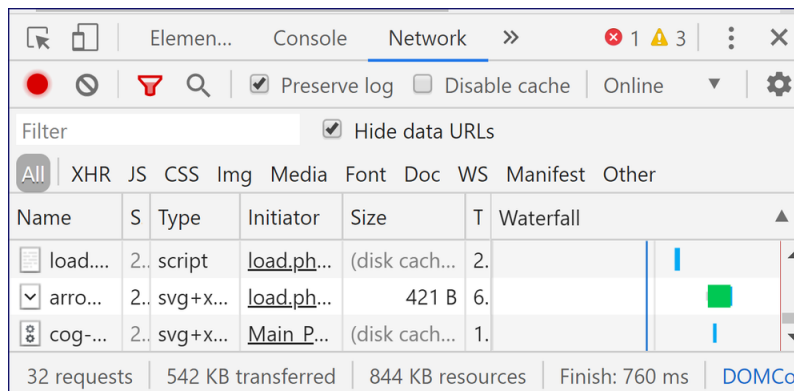
The `location` property of the browser is how the browser tracks the current web page being viewed. However, with JavaScript “single page” applications, the actual web page downloaded to the browser is always the same (`index.html` in our case), so we can use the `location` property to determine which components are mounted at any one time, without needing to download another file from the server.

First, let's take a look at how standard web pages work:

1. Visit <https://www.wikipedia.org/> in Google Chrome.
2. Open Google Chrome's **Network** tab and then clear it by pressing the Clear icon:



3. With the **Network** tab open, click any link on the page. You will see the Network tab fill up with downloaded assets:



4. Now open Vue/demo-viewer in VS Code's terminal by right-clicking the folder and selecting **Open in Integrated Terminal**.

5. Run `npm run serve` to launch the demo-viewer Vue 3 application.
6. Open and clear Google Chrome's **Network** tab.
7. Click around. Notice that the URL changes and the page display changes, but nothing gets downloaded. This is because each link click does not result in a fresh fetch of a web page from the web server. Instead, it results in a new router to a different component, which then updates the DOM.

EVALUATION COPY: Not to be used in class.



6.2. Vue Router

Vue Router is the official router for Vue 3. While it's not included in the core Vue 3 library's set of features, it is maintained and supported by same people who maintain and support Vue 3.

Implementing routes with Vue Router involves the following steps:

1. Define your routes.
2. Create a router instance.
3. Use the router instance.

Let's take a look at each of these step.

❖ 6.2.1. Defining routes

A route in Vue Router is a JavaScript object containing a path and a component that should be displayed when the browser's location matches that path. To define multiple routes, create an array of route objects. For example, in our app, we'll start with two routes: the one that displays when you first go to the app (the "" route) and the one that displays when the path is `/play`. Here's what the routes array should look like to enable these two routes:

```
const routes = [  
  { path: '/', component: GameConfig },  
  { path: '/play', component: Gameplay },  
];
```

You can also pass props to routes by adding dynamic parameters to the route. To add a parameter to a route, preface part of the path with a colon (:) and set the route's `props` option to `true`. In our game, we'll be passing the `operator` and `maxNumber` values selected on the configuration screen to the game play screen. Here's what the routes should look like with these parameters specified:

```
const routes = [
  { path: '/', component: GameConfig },
  { path: '/play/:operation/:maxNumber', component: GamePlay, props: true },
];
```

❖ 6.2.2. Creating a router instance

To create a router instance, pass an object containing a *history option* and the routes you defined into Vue Router's `createRouter()` function. The history option specifies how Vue Router should modify the browser URL. The options for the history mode are:

- Hash mode. Pass the hash mode option using `createWebHashHistory()`. Hash Mode adds a hash symbol (#) before the application's path. The hash symbol causes the browser to not refresh when a new path is loaded. Hash Mode doesn't require server configuration and is the mode that we'll be using.
- HTML5 Mode. HTML5 Mode is the recommended mode for live applications, but it requires a properly configured server. In HTML5 Mode, the URLs in the browser change without causing a page refresh.
- Memory Mode. Memory mode keeps route information in memory and doesn't rely on a browser. This mode is best for server-side applications and mobile apps.

❖ 6.2.3. Using a router instance

To make every component in your app able to use the router, you'll call Vue 3's `use()` function on your root component, like this:

```
app.use(router);
```

❖ 6.2.4. Displaying routes and linking

Once your routes are set up and made available to your components, the only thing left to do is to choose where the route components will display. To do this, put a `router-view` component in the template of the component where you want to display the route components.

To link between routes in your app, you use standard HTML a elements. For example, to make a link to the `/play` route, use the following HTML:

```
<a href="/play">Go to the play route</a>
```

Vue Router can also be controlled programatically by calling methods on the `this.$router` property. For example, to change the path to the `/play` route without the user clicking a link, you can use the following inside a component:

```
this.$router.push('/play');
```



Exercise 16: Implementing Routes

⌚ 15 to 25 minutes

In this exercise, we'll add routing to the Mathificent app.

1. Open `Exercises/mathficient` in the terminal.
2. Install **Vue Router** by running the following command:

```
npm install vue-router@4
```

3. Create the shells for two new single-file components in the `components` directory: `GameConfig` and `GamePlay`:

```
<template>
  GameConfig Component
</template>
```

```
<script>
export default {
  name: "GameConfig",
};
</script>
```

```
<template>
  GamePlay Component
</template>
```

```
<script>
export default {
  name: "GamePlay",
};
</script>
```

4. In `main.js`:
 - A. Import `createRouter` and `createWebHashHistory` from `vue-router` into `main.js`:

```
import { createRouter, createWebHashHistory } from 'vue-router';
```

- B. Import GameConfig and GamePlay into main.js:

```
import GameConfig from './components/GameConfig.vue';
import GamePlay from './components/GamePlay.vue';
```

- C. Create a routes array in main.js. The play route will take two parameters: operation and maxNumber. To make these two parameters available as props in GamePlay, set the props property to true.:

```
const routes = [
  { path: '/', component: GameConfig },
  { path: '/play/:operation/:maxNumber', component: GamePlay, props:true },
];
```

- D. Create a router using the routes array:

```
const router = createRouter({
  history: createWebHashHistory(),
  routes,
});
```

- E. Remove the existing statement that begins with createApp and replace it with the following three statements:

```
const app = createApp(App);
app.use(router);
app.mount('#app');
```

5. Open MainContainer and move the div element with the id of “config-container” into the template of GameConfig and delete its v-if directive.
6. Move the div element with the id of “game-container” from MainContainer into the template of GamePlay and delete its v-if directive. MainContainer should now only have an empty main tag with the id of “main-container” in its template.
7. Move the imports of SelectInput and PlayButton from MainContainer to the script element in GameConfig.
8. Move the imports of GameScore, GameTimer, GameEquation, and randInt to the script element in GamePlay component.

9. Add a components property to GameConfig and GamePlay with the appropriate imported components.
10. Put a routerview component inside the div in MainContainer's template:

```
<main id="main-container">
  <router-view></router-view>
</main>
```

11. Remove the components property and its contents from MainContainer.
12. Move the following data properties from MainContainer to GameConfig:

```
operations
operation
maxNumber
```

13. Move the following data properties from MainContainer to GamePlay:

```
buttons
input
operands
answered
score
gameLength
timeLeft
```

Evaluation
Copy

When you finish, the MainContainer should only have one data property: screen.

14. Delete the data property from MainContainer along with the screen property. In switching to using routes, we won't be needing that anymore.
15. Move the play() method from MainContainer to GameConfig.
16. Move all of the other methods from MainContainer to GamePlay.
17. Move the numbers computed property from MainContainer to GameConfig.
18. Move the question and equationClass computed properties from MainContainer to GamePlay.
19. Move all of the styles except the main-container style from MainContainer to GamePlay.

20. Add a props object to `GamePlay` to receive the props from the router.

```
props: {  
  operation: String,  
  maxNumber: String,  
},
```

21. Modify the `config()` in `GamePlay` to redirect the browser to the config route at `/`:

```
config() {  
  this.$router.push('/');  
},
```

22. Add the `mounted()` lifecycle method to `GamePlay` to start the game automatically when it's loaded:

```
mounted() {  
  this.newQuestion();  
  this.startTimer();  
},
```

23. Modify the `play()` method in `GameConfig` to redirect to the `play` route and pass the values of `operation` and `maxNumber` in the URL:

```
play() {  
  this.$router.push(  
    '/play/' + this.operation + '/' + this.maxNumber  
  );  
},
```

24. In the `play()` method, wrap `this.operation` with the `encodeURIComponent` function to encode the value of `this.operation` safely for URLs. This is necessary, because passing the symbol for division (`/`) in the URL would cause the browser to interpret it as part of the URL, rather than as part of the parameters passed in the URL.

```
play() {  
  this.$router.push(  
    '/play/' + encodeURIComponent(this.operation) + '/' + this.maxNumber  
  );  
},
```

25. Start up the application and test it out!

Congratulations! You now have routing set up!

Evaluation
Copy

Solution: Vue/Solutions/routing/main.js

```
1. import { createApp } from 'vue';
2. import { createRouter, createWebHashHistory } from 'vue-router';
3. import App from './App.vue';
4. import GameConfig from './components/GameConfig';
5. import GamePlay from './components/GamePlay';
6.
7. const routes = [
8.   { path: '/', component: GameConfig },
9.   { path: '/play/:operation/:maxNumber', component: GamePlay, props: true },
10. ];
11.
12. const router = createRouter({
13.   history: createWebHashHistory(),
14.   routes,
15. });
16.
17. const app = createApp(App);
18. app.use(router);
19. app.mount('#app');
```

Solution: Vue/Solutions/routing/MainContainer.vue

```
1. <template>
2.   <main id="main-container">
3.     <router-view></router-view>
4.   </main>
5. </template>
6.
7. <script>
8. export default {
9.   name: 'MainContainer',
10.  components: {},
11. };
12. </script>
13.
14. <style scoped>
15. #main-container {
16.   margin: auto;
17.   width: 380px;
18. }
19. </style>
```

Solution: Vue/Solutions/routing/GameConfig.vue

```
1.   <template>
2.     <div id="config-container">
3.       <h1>Mathificent</h1>
4.       <SelectInput
5.         :currentValue="operation"
6.         label="Operation"
7.         id="operation"
8.         v-model="operation"
9.         :options="operations"
10.        @input="(o) => (this.operation = o)"
11.      />
12.      <SelectInput
13.        :currentValue="maxNumber"
14.        label="Maximum Number"
15.        id="max-number"
16.        v-model="maxNumber"
17.        :options="numbers"
18.        @input="(n) => (this.maxNumber = n)"
19.      />
20.      <PlayButton @play-button-click="play" />
21.    </div>
22.  </template>
23.
24.  <script>
25.    import SelectInput from './SelectInput';
26.    import PlayButton from './PlayButton';
27.    export default {
28.      name: 'GameConfig',
29.      components: {
30.        SelectInput,
31.        PlayButton,
32.      },
33.      data: function () {
34.        return {
35.          operations: [
36.            ['Addition', '+'],
37.            ['Subtraction', '-'],
38.            ['Multiplication', 'x'],
39.            ['Division', '/'],
40.          ],
41.          operation: 'x',
42.          maxNumber: '10',
43.        };
44.      },
```

Evaluation
Copy

```
45.     methods: {
46.       play() {
47.         this.$router.push(
48.           '/play/' + encodeURIComponent(this.operation) + '/' + this.maxNumber
49.         );
50.       },
51.     },
52.     computed: {
53.       numbers: function () {
54.         const numbers = [];
55.         for (let number = 2; number <= 100; number++) {
56.           numbers.push([number, number]);
57.         }
58.         return numbers;
59.       },
60.     },
61.   };
62. </script>
```

Solution: Vue/Solutions/routing/GamePlay.vue

```
1. <template>
2.   <div id="game-container" class="text-center">
3.     <transition name="slide">
4.       <template v-if="timeLeft === 0">
5.         <div>
6.           <h2>Time's Up!</h2>
7.           <strong class="big">You Answered</strong>
8.           <div class="huge">{{ score }}</div>
9.           <strong class="big">Questions Correctly</strong>
10.          <button
11.            class="btn btn-primary form-control m-1"
12.            v-on:click="restart()"
13.          >
14.            Play Again with Same Settings
15.          </button>
16.          <button
17.            class="btn btn-secondary form-control m-1"
18.            v-on:click="config()"
19.          >
20.            Change Settings
21.          </button>
22.        </div>
23.      </template>
24.    </transition>
25.    <transition name="slide-right">
26.      <template v-if="timeLeft > 0">
27.        <div>
28.          <div class="row border-bottom" id="scoreboard">
29.            <div class="col px-3 text-left">
30.              <GameScore :score="score" />
31.            </div>
32.            <div class="col px-3 text-right">
33.              <GameTimer :timeLeft="timeLeft" />
34.            </div>
35.          </div>
36.          <div :class="equationClass" id="equation">
37.            <GameEquation
38.              :question="question"
39.              :answer="input"
40.              :answered="answered"
41.            />
42.          </div>
43.          <div class="row" id="buttons">
44.            <div class="col">
```

Evaluation
Copy

```

45.         <button
46.             class="btn btn-primary number-button"
47.             v-for="button in buttons"
48.             :key="button"
49.             @click="setInput(button)"
50.         >
51.             {{ button }}
52.         </button>
53.         <button class="btn btn-primary" id="clear-button" @click="clear">
54.             Clear
55.         </button>
56.     </div>
57. </div>
58. </div>
59. </template>
60. </transition>
61. </div>
62. </template>
63. <script>
64. import GameScore from './GameScore';
65. import GameTimer from './GameTimer';
66. import GameEquation from './GameEquation';
67. import { randInt } from '../helpers/helpers';
68. export default {
69.     name: 'GamePlay',
70.     components: {
71.         GameScore,
72.         GameTimer,
73.         GameEquation,
74.     },
75.     data: function () {
76.         return {
77.             buttons: [1, 2, 3, 4, 5, 6, 7, 8, 9, 0],
78.             input: '',
79.             operands: { num1: '1', num2: '1' },
80.             answered: false,
81.             score: 0,
82.             gameLength: 60,
83.             timeLeft: 0,
84.         };
85.     },
86.     props: {
87.         operation: String,
88.         maxNumber: String,
89.     },

```

```

90.     methods: {
91.         config() {
92.             this.$router.push('/');
93.         },
94.
95.         setInput(value) {
96.             this.input += String(value);
97.             this.input = String(Number(this.input));
98.             this.answered = this.checkAnswer(
99.                 this.input,
100.                 this.operation,
101.                 this.operands
102.             );
103.             if (this.answered) {
104.                 setTimeout(this.newQuestion, 300);
105.                 this.score++;
106.             }
107.         },
108.         clear() {
109.             this.input = '';
110.         },
111.         getRandNumbers(operator, low, high) {
112.             let num1 = randInt(low, high);
113.             let num2 = randInt(low, high);
114.             const numHigh = Math.max(num1, num2);
115.             const numLow = Math.min(num1, num2);
116.
117.             if (operator === '-') {
118.                 // Make sure higher num comes first
119.                 num1 = numHigh;
120.                 num2 = numLow;
121.             }
122.
123.             if (operator === '/') {
124.                 if (num2 === 0) {
125.                     // No division by zero
126.                     num2 = randInt(1, high);
127.                 }
128.                 num1 = num1 * num2;
129.             }
130.             return { num1, num2 };
131.         },
132.         checkAnswer(userAnswer, operation, operands) {
133.             if (isNaN(userAnswer)) return false; // User hasn't answered
134.

```

```

135.     let correctAnswer;
136.     switch (operation) {
137.       case '+':
138.         correctAnswer = operands.num1 + operands.num2;
139.         break;
140.       case '-':
141.         correctAnswer = operands.num1 - operands.num2;
142.         break;
143.       case 'x':
144.         correctAnswer = operands.num1 * operands.num2;
145.         break;
146.       default: // division
147.         correctAnswer = operands.num1 / operands.num2;
148.     }
149.     return parseInt(userAnswer) === correctAnswer;
150.   },
151.   newQuestion() {
152.     this.input = '';
153.     this.answered = false;
154.     this.operands = this.getRandNumbers(this.operation, 0, this.maxNumber);
155.   },
156.   startTimer() {
157.     window.addEventListener('keyup', this.handleKeyUp);
158.     this.timeLeft = this.gameLength;
159.     if (this.timeLeft > 0) {
160.       this.timer = setInterval(() => {
161.         this.timeLeft--;
162.         if (this.timeLeft === 0) {
163.           clearInterval(this.timer);
164.           window.removeEventListener('keyup', this.handleKeyUp);
165.         }
166.       }, 1000);
167.     }
168.   },
169.   restart() {
170.     this.score = 0;
171.     this.startTimer();
172.     this.newQuestion();
173.   },
174.   handleKeyUp(e) {
175.     e.preventDefault(); // prevent the normal behavior of the key
176.     if (e.keyCode === 32 || e.keyCode === 13) {
177.       // space/Enter
178.       this.clear();
179.     } else if (e.keyCode === 8) {

```

```

180.         // backspace
181.         this.input = this.input.substring(0, this.input.length - 1);
182.     } else if (!isNaN(e.key)) {
183.         this.setInput(e.key);
184.     }
185. },
186. },
187. mounted() {
188.     this.newQuestion();
189.     this.startTimer();
190. },
191. computed: {
192.     question: function () {
193.         const num1 = this.operands.num1;
194.         const num2 = this.operands.num2;
195.         const equation = `${num1} ${this.operation} ${num2}`;
196.         return equation;
197.     },
198.     equationClass: function () {
199.         if (this.answered) {
200.             return 'row text-primary my-2 fade';
201.         } else {
202.             return 'row text-secondary my-2';
203.         }
204.     },
205. },
206. };
207. </script>
208. <style scoped>
209. button.number-button {
210.     border-radius: 0.25em;
211.     font-size: 3em;
212.     height: 2em;
213.     margin: 0.1em;
214.     text-align: center;
215.     width: 2em;
216. }
217.
218. #clear-button {
219.     border-radius: 0.25em;
220.     font-size: 3em;
221.     height: 2em;
222.     margin: 0.1em;
223.     text-align: center;
224.     width: 4.2em;

```

Evaluation
Copy

```
225. }
226.
227. #scoreboard {
228.   font-size: 1.5em;
229. }
230.
231. .big {
232.   font-size: 1.5em;
233. }
234.
235. .huge {
236.   font-size: 5em;
237. }
238.
239. .slide-leave-active,
240. .slide-enter-active {
241.   position: absolute;
242.   top: 56px;
243.   transition: 1s;
244.   width: 380px;
245. }
246.
247. .slide-enter-from {
248.   transform: translate(-100%, 0);
249.   transition: opacity 0.5s;
250. }
251.
252. .slide-leave-to {
253.   opacity: 0;
254.   transform: translate(100%, 0);
255. }
256.
257. .slide-right-leave-active,
258. .slide-right-enter-active {
259.   position: absolute;
260.   top: 56px;
261.   transition: 1s;
262.   width: 380px;
263. }
264.
265. .slide-right-enter-from {
266.   transform: translate(100%, 0);
267.   transition: opacity 0.5s;
268. }
269.
```



```
270. .slide-right-leave-to {  
271.   opacity: 0;  
272.   transform: translate(-100%, 0);  
273. }  
274. </style>
```

Evaluation
Copy

Conclusion

In this lesson, you have learned about routing and how Vue Router makes it possible to change the component being displayed based on the browser's location property.