

Intro to Vue.js



Week 11 Overview

Monday

Intro to Vue.js

Tuesday

Matchmaking
(*no class*)

Wednesday

Vue Events

Thursday

Components
& Vuex

Friday

Review

Today's Objectives

1. Introduction to Vue

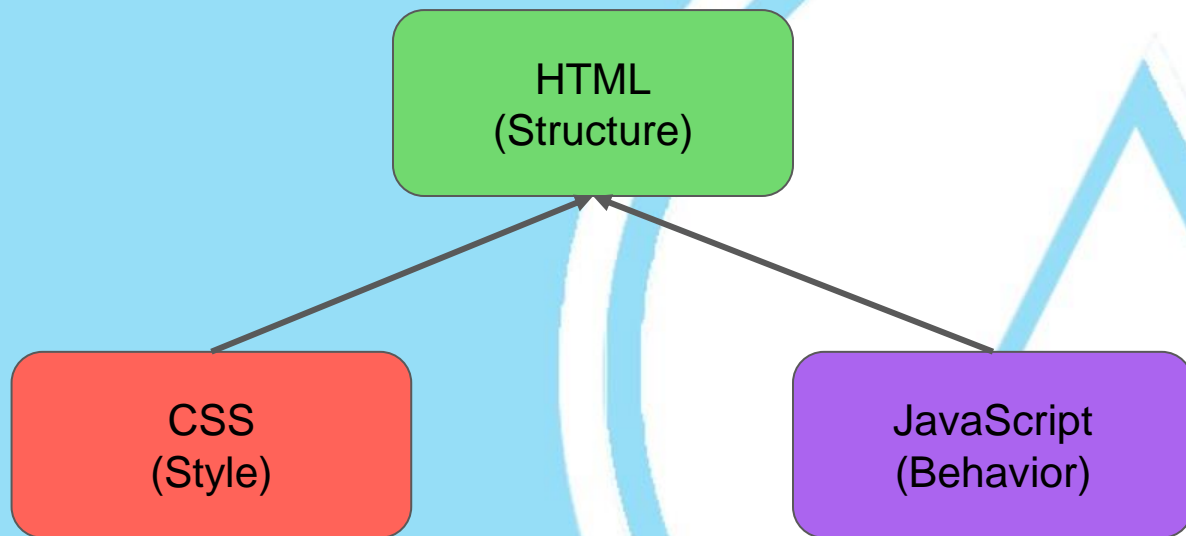
- a. Creating a Vue Project
- b. Single Page Applications (SPA)
- c. Parts of a Vue.js project
- d. Components

2. Vue Data Binding

- a. 1-Way Data Binding
- b. Arrays with **v-for**
- c. Binding DOM Element Attributes with **v-bind**
- d. Computed Properties
- e. Conditional Elements using **v-if**
- f. 2-Way Data Binding with **v-model**
- g. Conditional Attributes with **v-bind**

Motivation for JavaScript Frameworks

- Traditionally, web pages are broken up into separate files:



Motivation for JavaScript Frameworks

- Complex sites led to unmanageable sizes & repeated code
- Solution: multiple files
 - **CSS**
 - *Examples:*
 - `header.css`
 - `footer.css`
 - **JavaScript**
 - *Examples*
 - `eventHandlers.js`
 - `apiCalls.js`
- As frontend functionality increased, shift toward more modular model of structured languages seemed necessary



*JavaScript
frameworks to
the rescue!!!!*

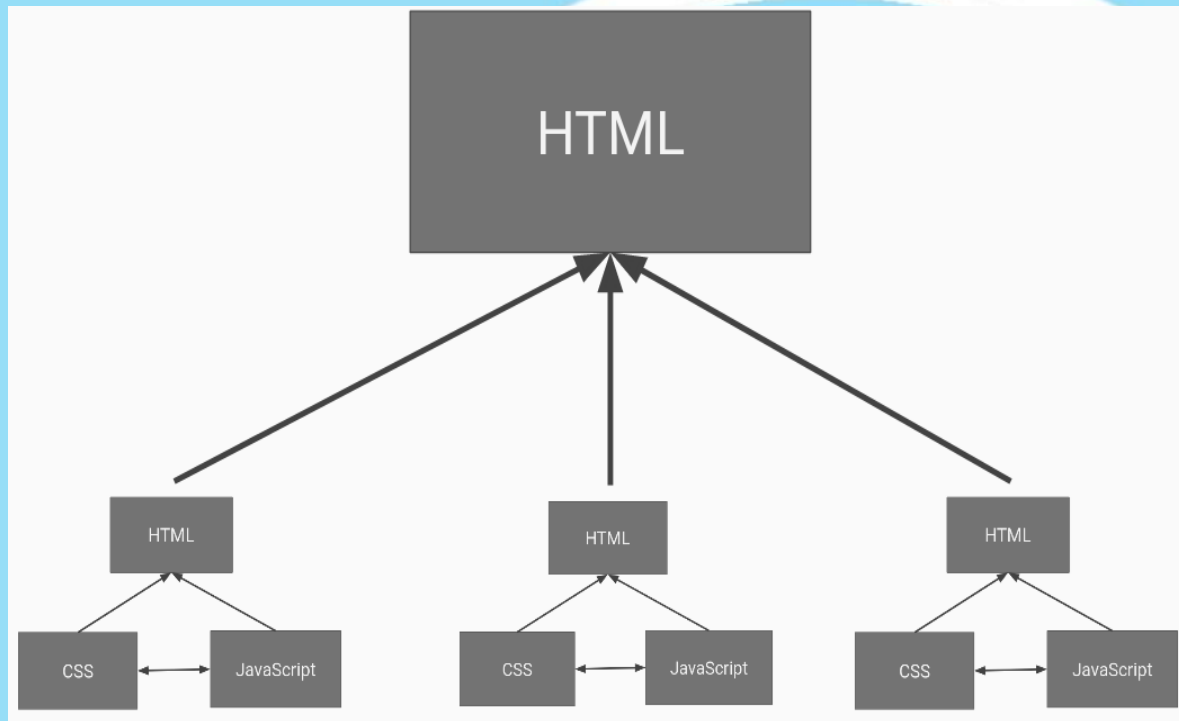
Image by storyset on [Freepik](#)

Motivation for JavaScript Frameworks

- JavaScript frameworks:
 - Vue.js
 - React
 - Angular
- Implement encapsulation in JavaScript
 - Build reusable **components**
 - know how to display themselves
 - contain behavior logic
 - have internal state data much like Java objects
 - Assemble pages from components

JavaScript Components

- HTML page
 - Main content
 - Structure
- Components
 - HTML
 - CSS
 - JavaScript
 - Act as single "plug-and-play" front-end element.
 - Simplify creation and maintenance



Introducing...



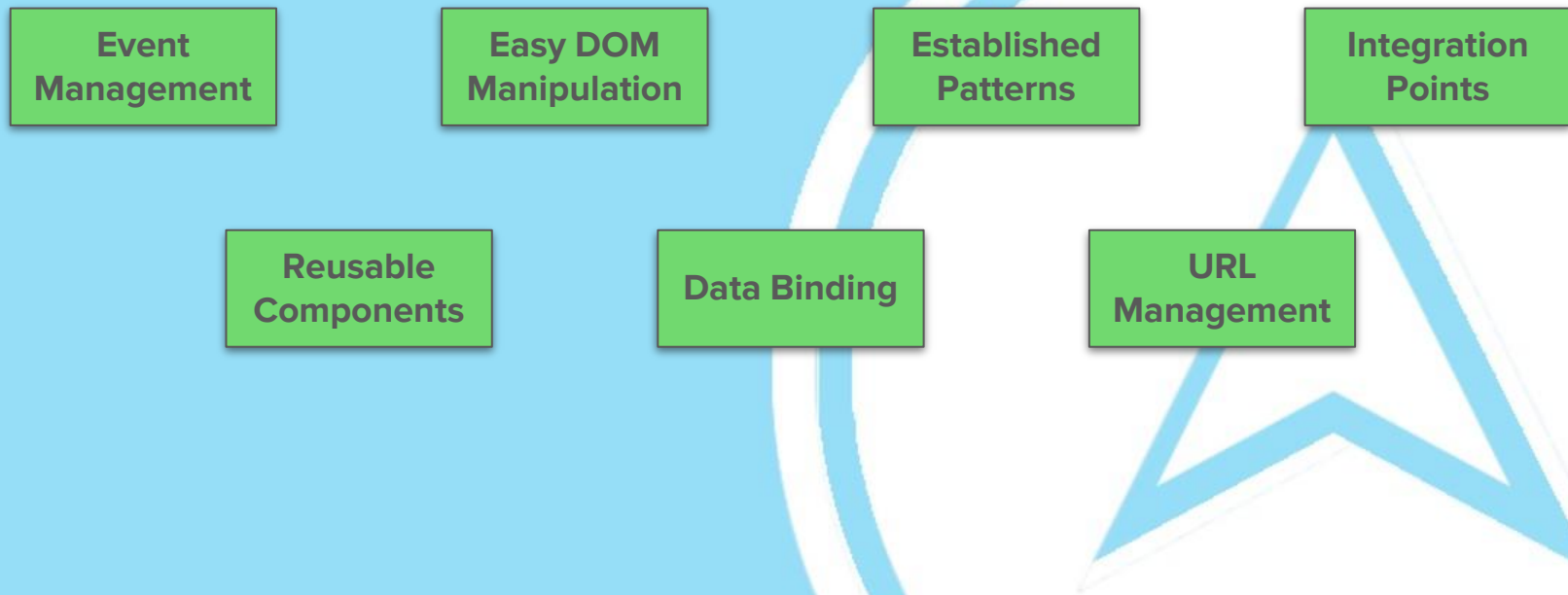
Vue.js

- Open source competitor to frameworks like Angular (Google) and React (Facebook)

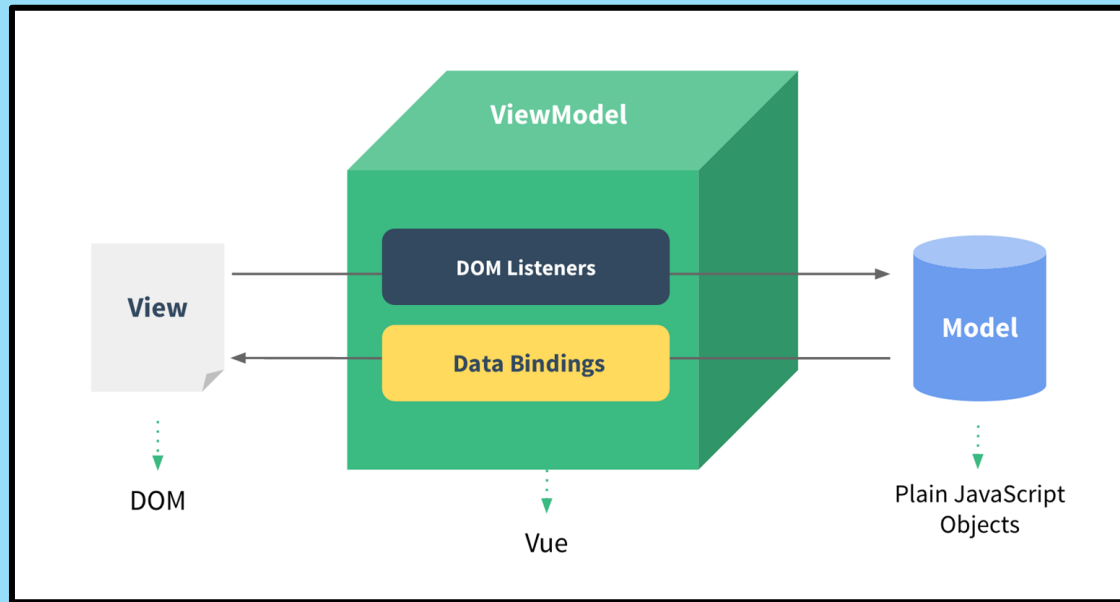


- Concepts in Vue.js should map well to other frameworks

Vue.js is a JavaScript Framework

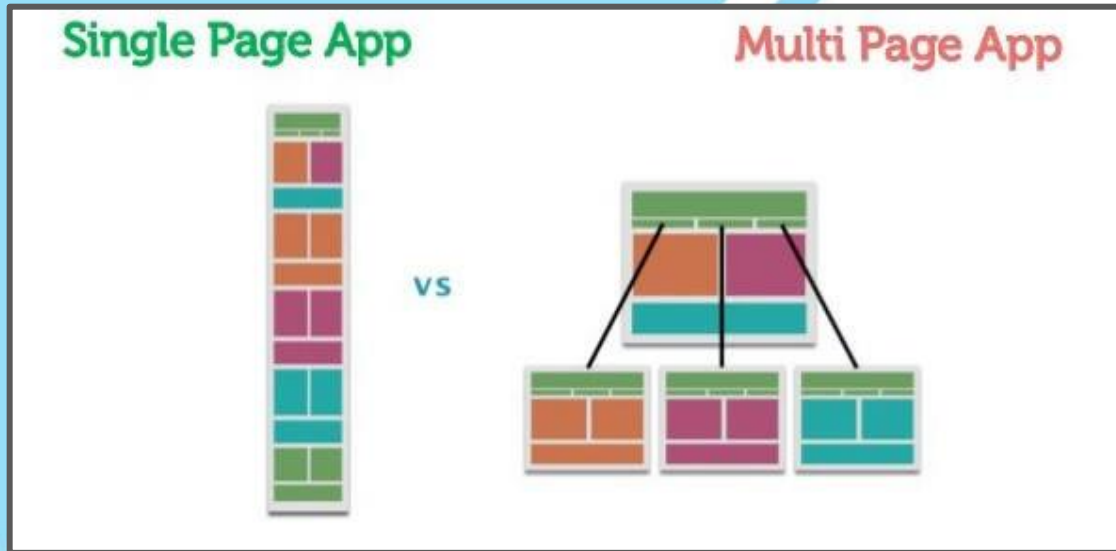


Vue.js Design



Single Page Application (SPA)

- **Single Page Applications** serve a single HTML page
 - Do not follow the old model of serving different HTML files for each page
 - All “pages” generated using DOM manipulation of the original HTML with new data loaded using JavaScript.



Single Page Application (SPA)

Pros

- Better user experience due to instant responsive feedback
- Servers require less resources
- Allows more reusable Server Side Code, the server uses web services to provide data, so it is further decoupled from the view
- Using caching can be transformed into an offline experience (e.g. Google Drive)
- Allows separation of focus for developers: frontend developer can focus fully on the frontend and backend developers fully on the backend

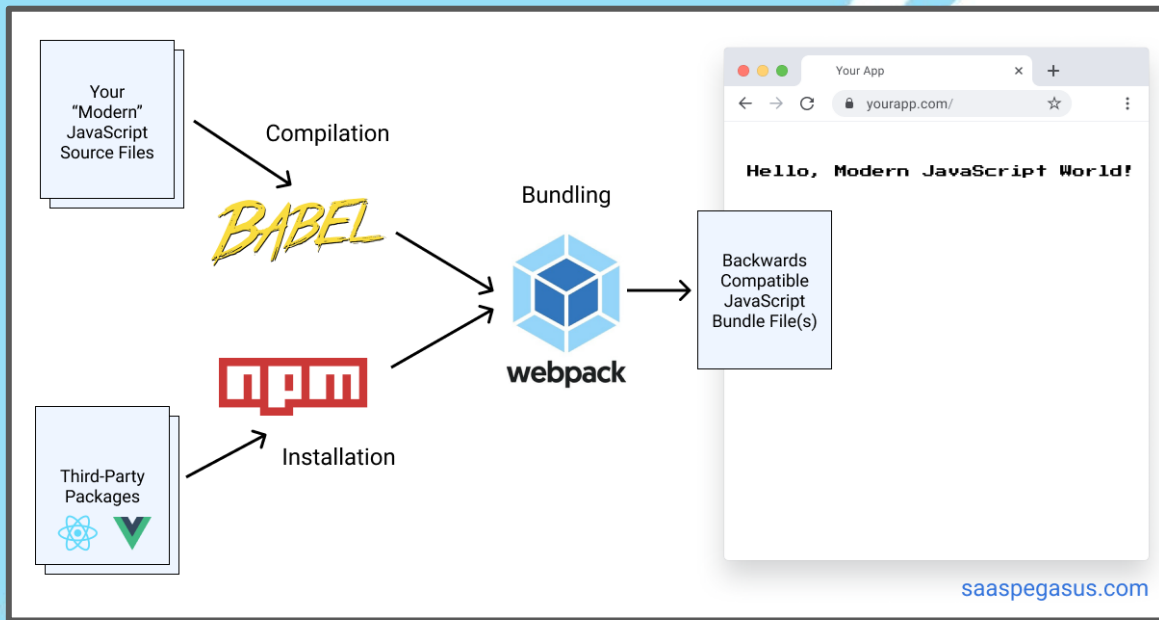
Single Page Application (SPA)

Cons

- Slower upfront load time
- Work performed on the client and often kept open for hours rather than minutes like a traditional web page, developers must be aware of possible code issues that do not affect traditional web pages, like memory leaks.
- Search engines cannot effectively index them, so sites that rely on content are not effective as a SPA (e.g. a Blog)
- Requires more backend development
- Limits users without JavaScript, slow machines, or Accessibility Needs. (e.g. Screen Readers).
- Browsers default navigation (Back button, etc) does not work, so SPA must handle these events.
 - The browser's History API and Routers help deal with this

The Vue.js Toolchain

- Set of programming tools that work together to perform a complex task or create a software product by linking together sequential to each provide part of the needed functionality.





Node Package Manager (NPM)

- Manages Dependencies and builds
- Similar to Maven in Java



JavaScript **Compiler** and **Transpiler** that provides **Polyfill**

- **Compile** - compiles other languages, like TypeScript, to JavaScript
- **Transpile** - converts modern JavaScript syntax (ECMAScript 6), which is not yet fully supported by browsers, into older style JavaScript that those browser can use.
- **Polyfill** - fills gaps in support for newer web features, like HTML5, by providing functionality in a way that devices that do not support those newer features do support.

Creating a new Vue.js Project



- Vue CLI

- Command Line Interface
- Simplifies setting up a project and creating new components

- **vue create**

- Sets up everything for a new project
- Allows you to choose options for project
- For today:
 - **vue create vue-product-reviews**
 - Select **Default ([Vue 2] babel, eslint)** as project type

- **npm run serve**

- Starts server
- Must be run in same directory as **package.json**

Parts of a Vue Project

vue-product-reviews

node_modules

public

★ favicon.ico

<> index.html

src

assets

components

▼ App.vue

JS main.js

📄 .gitignore

JS babel.config.js

{ } package-lock.json

{ } package.json

📖 README.md

1. node_modules

- a. JavaScript libraries required to run project
- b. Created when **npm install** is run for a project

2. public

- a. Initial HTML file and other static files

3. src

- a. Vue.js source code - where most Vue.js application development is done
- b. Sub-folders
 - i. **assets** - images, etc.
 - ii. **components** - Vue.js components (**.vue** files)
- c. **App.vue**
 - i. Starting component
 - ii. Good place to add global components (i.e. Header), global CSS, etc.
 - 1. Starting component does not need to be **App.vue** (is by default)
- d. **main.js** - Single Vue.js instance that starts Vue.js running and loads the initially components (like Java **main** method)

Parts of a Vue.js Project - Everything Else

1. **.gitignore**: sensible defaults for .gitignore
2. **babel.config.js**: babel configuration
3. **package-lock.json**: It describes the exact tree that was generated, such that subsequent installs are able to generate identical trees, regardless of intermediate dependency updates.
4. **package.json**: The NPM package meta file that contains all the build dependencies and build commands.
5. **README.md**: default readme file

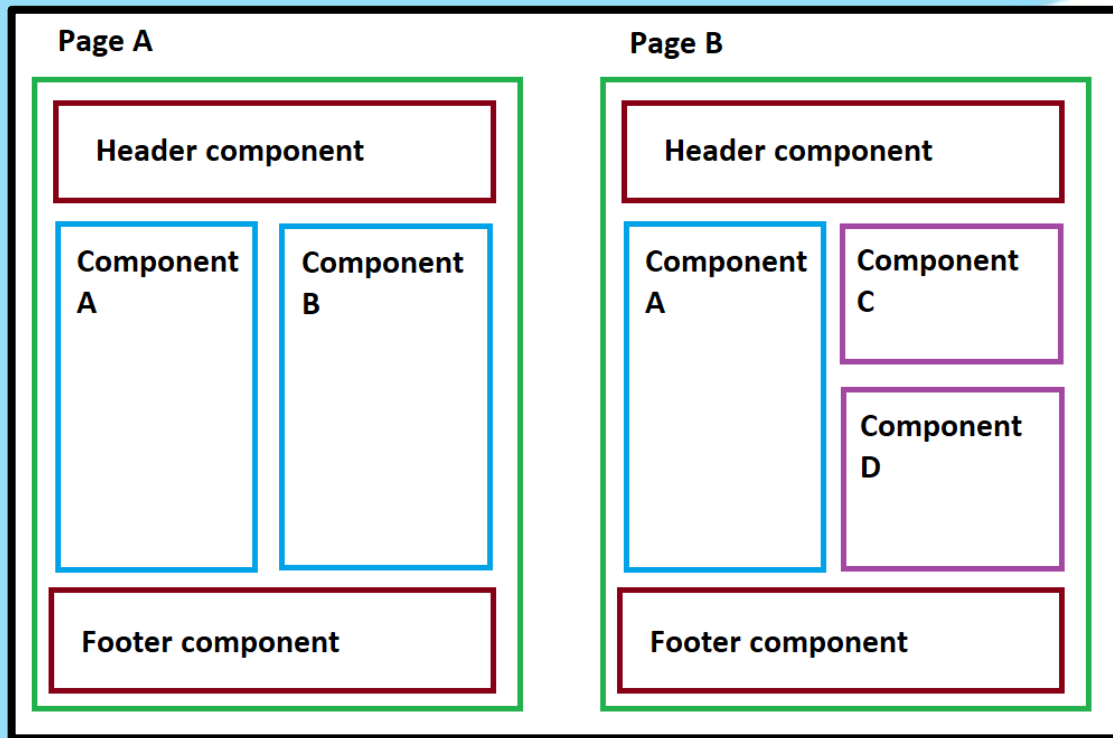
package.json

```
1 {
2   "name": "your-project-name",
3   "version": "0.1.0",
4   "private": true,
5   <!-- anything in scripts can be run via npm run name-of-script -->
6   "scripts": {
7     "serve": "vue-cli-service serve",
8     "build": "vue-cli-service build",
9     "lint": "vue-cli-service lint",
10    "test:unit": "vue-cli-service test:unit"
11  },
12  <!-- These are included with production builds -->
13  "dependencies": {
14    "core-js": "^3.4.4",
15    "vue": "^2.6.10"
16  },
17  <!-- These are only needed for development and building files -->
18  "devDependencies": {
19    "@vue/cli-plugin-babel": "^4.1.0",
20    "@vue/cli-plugin-eslint": "^4.1.0",
21    "@vue/cli-plugin-unit-mocha": "^4.1.2",
22    "Even more omitted...": "0.1.1"
23  },
24  "otherThingsMayGoHere": {}
25 }
26
```

Vue.js Components



Vue Components



Anatomy of a Vue.js Component

```
<template>
  <div class="main">
    <h2>Product Reviews for {{ name }}</h2>
    <p class="description">{{ description }}</p>
  </div>
</template>

<script>
export default {
  name: 'product-review',
  data() {
    return {
      name: 'Cigar Parties for Dummies',
      description: 'Host and plan the perfect cigar party for all of y
    }
  }
}
</script>

<style scoped>
div.main {
  margin: 1rem 0;
}
</style>
```

- Vue.js components made up of
 - **<template>** section
 - Contains HTML elements
 - **<script>** section
 - Contains JavaScript code
 - **<style>** section
 - Contains CSS styling
- What user sees on screen is defined mostly by
 - HTML elements in **<template>** section
 - CSS styles applied in the **<style>** section
- Behavior of component defined by what's in **<script>** section

Anatomy of a Vue.js Component

```
<template>
  <div class="main">
    <h2>Product Reviews for {{ name }}</h2>

    <p class="description">{{ description }}</p>
  </div>
</template>

<script>
export default {
  name: 'product-review',
  data() {
    return {
      name: 'Cigar Parties for Dummies',
      description: 'Host and plan the perfect cigar party for all of y
    }
  }
}
</script>

<style scoped>
div.main {
  margin: 1rem 0;
}
</style>
```

- **<template>** section must contain a **single** root element (**div**, **section**, **main**, etc.) to contain all elements of the component

- **<script>** section
 - Contains JavaScript
 - Mostly inside **export default {}** block
 - **data()** must return object of key/value pairs for data page uses

- **scoped** attribute declares the CSS in the **<style>** section only affects **THIS** component.
- If **scoped** attribute is omitted, styles will apply to selectors that match these CSS rules in **ANY** component

Adding a Component to Another Component

- In order to use a component in another component you must include the new component in **three** places.

```
<template>
  <div id="app">
    <product-review />
  </div>
</template>

<script>
import ProductReview from './components/ProductReview.vue';

export default {
  name: "app",
  data() {
    return {

    }
  },
  components: {
    ProductReview
  }
}
</script>
```

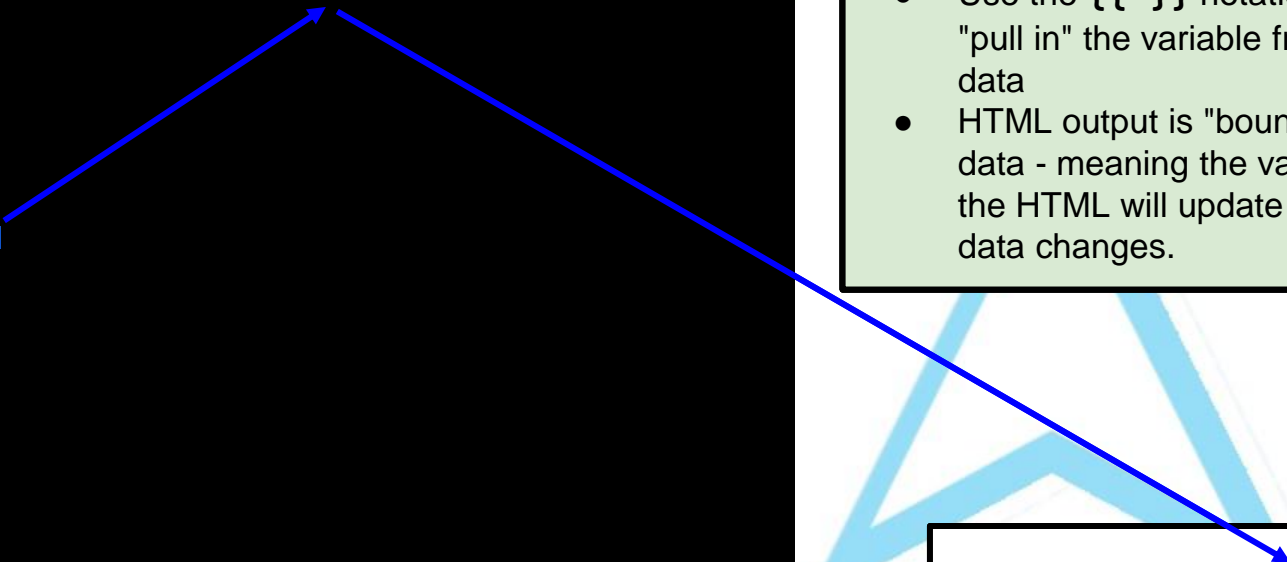
- Use component as tag in **<template>** section
 - HTML tags usually snake case
 - Vue.js accepts snake OR camel/pascal case
 - Import component file in **<script>** section
 - List component in **components** section of **<script>** section
- Variable name after **import** and name listed in **components** section must match

1 Way Data Binding

```
<template>
  <div class="main-section">
    <h1>Hello! My name is <span class="name-text">{{ name }}</span></h1>
  </div>
</template>

<script>
export default {
  data() {
    return {
      name: 'Linda'
    }
  }
}
</script>

<style>
.name-text {
  color: blue;
}
</style>
```



- Can output a value from the component's data in the HTML
- Use the `{{ }}` notation to "pull in" the variable from the data
- HTML output is "bound" to the data - meaning the value in the HTML will update if the data changes.

Hello! My name is Linda

Looping with **v-for**

- **v-for** allows you to loop through a set of data and create an element for each item in the data set
- For instance, a **div** element with a **v-for** loop iterating through 5 users would generate 5 **div** elements (one for each user)
- In addition to the **v-for** directive, an element using **v-for** should have a unique key to uniquely identify each row (more on this soon)
- Example (we'll dig into this in the next slide):

```
users: [  
  {id: 1, name: 'Yoav'},  
  {id: 2, name: 'Linda'},  
  {id: 3, name: 'Cindy'},  
]
```



```
<div v-for="user in users" v-bind:key="user.id">  
  User {{ user.id }}: {{ user.name }}  
</div>
```



User 1: Yoav
User 2: Linda
User 3: Cindy

Looping with **v-for**

users is an array of user objects

```
users: [  
  {id: 1, name: 'Yoav'},  
  {id: 2, name: 'Linda'},  
  {id: 3, name: 'Cindy'},  
]
```

v-for directive

iterates through **users** array, pulling out one element at a time into a **user** variable (think Java **for-each** loop)

uses the **id** of the current **user** as the key (we'll learn what **v-bind:** is shortly)

```
<div v-for="user in users" v-bind:key="user.id">  
  User {{ user.id }}: {{ user.name }}  
</div>
```

the **v-for** will create 3 **div** elements using the values in the current **user** to generate the text

User {{ user.id }}: {{ user.name }}

```
<div>  
  User 1: Yoav  
</div>  
<div>  
  User 2: Linda  
</div>  
<div>  
  User 3: Cindy  
</div>
```

the HTML output will look like this

User 1: Yoav
User 2: Linda
User 3: Cindy

Using the Array Index as the Key

- Can add a second parameter to the **v-for** clause to capture the array index
- Param can be called anything – just gets index because it is a second param
- Can use this param as the key
- Can use this if you need to know the actual array index as you are iterating

```
<div v-for="(user, indexVal) in users" v-bind:key="indexVal">  
  User {{ user.id }} (Index: {{ indexVal }}): {{ user.name }}  
</div>
```

Second param to capture the index.
(Note you need parantheses around the params if there is more than one.)

The index value can also be output or used in logic in the HTML.

Since the index value is unique, it can be used as the key value.

Binding to DOM element Attributes with **v-bind**

- Prefixing an attribute on a DOM element with **v-bind**: allows the attribute to be evaluated as an expression rather than being treated as text.
 - This allows you to pass in values or objects based on your internal data or even use simple logic evaluation to determine the value
- Let's look at our previous example:

```
<div v-for="user in users" v-bind:key="user.id">  
  User {{ user.id }}: {{ user.name }}  
</div>
```

Because **v-bind**: is used frequently, there is shorthand for it. You can leave off the **v-bind** and just use **:** to indicate **v-bind**:

*Example: **v-bind:key** can be written as **:key***

Prefixing **key** with **v-bind**: means that the value of key will be the result of evaluating the expression `user.id`.

So if the id of the current user is 2, the value of key will be 2.

Without the **v-bind**: prefix, the value **key** would be the actual **TEXT** `user.id` **NOT** the actual **id** of the user (2)

Computed Properties

- Allows one way data binding to a calculated value
- Similar to a derived property in Java
- Looks like a function but seen by the code as property (more on this tomorrow)
- If the value of the computed property changes then the bound text is updated on the page

Number of users: 4

If another user is added the number updates to 4

```
<template>
  <div class="user-container">
    <h3>Number of users: {{ numOfUsers }}</h3>
  </div>
</teplate>

<script>
export default {
  data() {
    return {
      users: [
        { id: 1, name: "Yoav" },
        { id: 2, name: "Linda" },
        { id: 3, name: "Cindy" },
      ],
    };
  },
  computed: {
    numOfUsers() {
      return this.users.length;
    },
  },
};
</script>
```

2-Way Model Binding with `v-model`

```
<template>
  <div>
    <form>
      <label for="name">Name: </label>
      <input id="name" type="text" v-model="name"/>
    </form>
  </div>
</template>

<script>
export default {
  data() {
    return {
      name: ''
    }
  }
}
</script>
```

A blue arrow originates from the `name: ''` property in the `data()` function and points to the `v-model="name"` attribute in the `<input>` tag, illustrating the binding between the form element and the data model.

- Form elements, which can be changed by a user (text fields, radio buttons, select boxes, etc.), can be bound using 2-way data binding using the **`v-model`** attribute
- 2-way data binding keeps the data and the user input in sync
 - When form element is changed by the user, data is reactively updated
 - If the data is changed the form element is reactively updated

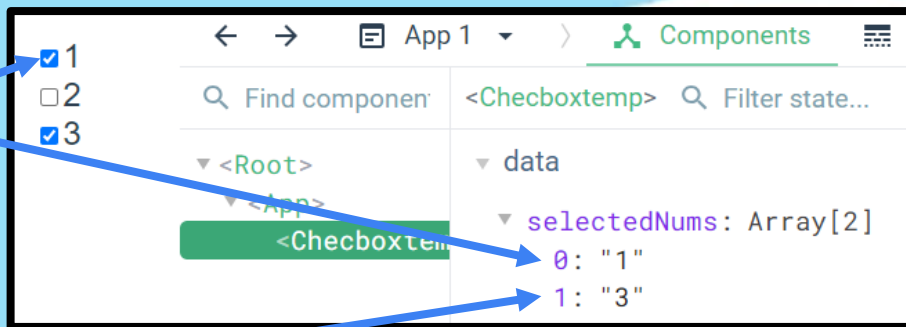
v-model modifiers

- **v-model.number**
 - Converts the input to a number when updating the bound property
- **v-model.trim**
 - Trims any whitespace at the left / right of the string
- **v-model.lazy**
 - Waits for the input element to lose focus before updating the bound property

v-model with Checkboxes

```
<template>
  <div class="box-demo">
    <div>
      <input id="one" type="checkbox" value="1"
        v-model="selectedNums" />
      <label for="one">1</label>
    </div>
    <div>
      <input id="two" type="checkbox" value="2"
        v-model="selectedNums" />
      <label for="two">2</label>
    </div>
    <div>
      <input id="three" type="checkbox" value="3"
        v-model="selectedNums" />
      <label for="three">3</label>
    </div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      selectedNums: []
    }
  }
}
</script>
```



- You can bind several checkboxes with different values to the same array property in your data section using **v-model**
- Using **v-model** this way causes values of checkboxes bound to the array to get added or removed from the array as the boxes are checked or unchecked

Conditional Elements: **v-if** and **v-show**

- Vue.js allows you to control whether or not elements are displayed based on a boolean expression
- Adding **v-if** to an element will
 - Display the element if the corresponding expression evaluates to **true**
 - Remove the element from the DOM if the corresponding expression evaluates to **false**
- Adding **v-show** to an element will
 - Display the element if the corresponding expression evaluates to **true**
 - Hide the element if the corresponding expression evaluates to **false**
 - Does **NOT** remove the element from the DOM, hides (**display: none**)
 - Manipulating the DOM is far less efficient than hiding or showing an element using CSS so unless you have a good reason to use **v-if**, use **v-show**

```
<div v-show="name === ' ' ">
  Please enter info above
</div>
```

Using `v-bind:class` to Dynamically Apply Styles

- Vue.js allows you to dynamically apply styles to an element using `v-bind:class` dynamically

```
<template>
  <div>
    <form>
      <label for="name"
        v-bind:class="{ redtext: name === '', italictext: name === ''}">
        Name: </label>
      <input id="name" type="text" v-model="name"
        placeholder="Enter a name"
        :class="{ 'border-red': name === '' }"/>
    </form>
  </div>
</template>
```

Note
shorthand
for `v-
bind:`

If style
name has
a dash in
it, must be
in quotes

`v-bind:class` value
is an object (`{ }`)

properties are
style names

- values are boolean expressions
- if evaluates to **true**, style listed as property will be applied
- if evaluates to **false**, style listed as property will NOT be applied

Code Walkthrough

