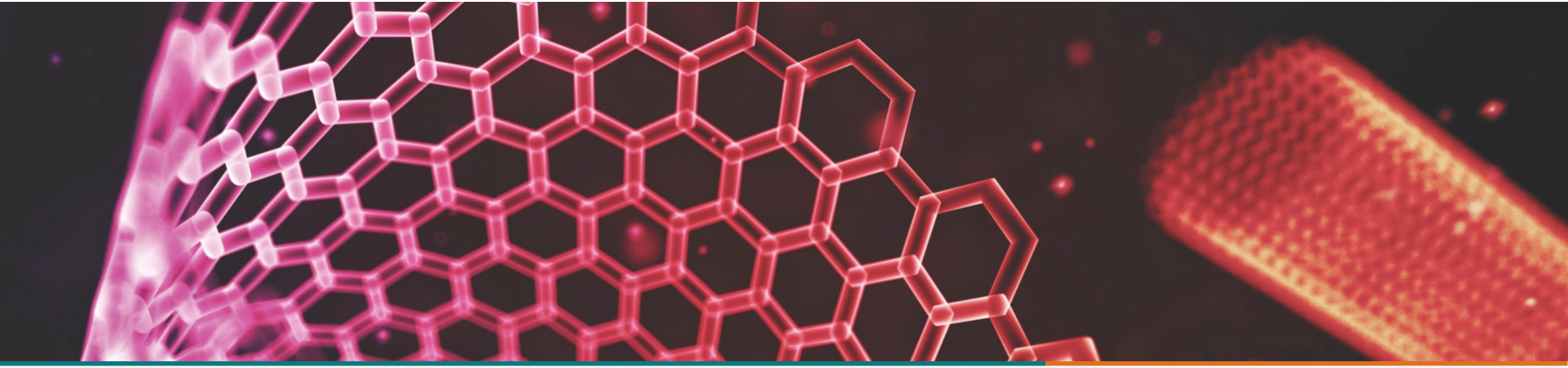
STEVENS
INSTITUTE of TECHNOLOGY
Schaefer School of Engineering & Science

CS 554 – Web Programming II

Vue.JS



STEVENS

INSTITUTE of TECHNOLOGY

Schaefer School of Engineering & Science

stevens.edu

Patrick Hill

Adjunct Professor

Computer Science Department

Patrick.Hill@stevens.edu

What is Vue?

Vue is a JavaScript Framework for building user interfaces.

Like we have seen in React, Vue allows you to create reusable components each with their own HTML, logic, and CSS needed to render that part of the page.

Installing Vue

You can install Vue a few ways.

- Using a CDN Script tag
  - `<script src="https://cdn.jsdelivr.net/npm/vue@2.6.11"></script>`
- Using the Vue CLI (Similar to create-react-app)
  - **npm install –g @vue/cli**
  - Create a new Vue application by typing **vue create my-app** into the terminal
- NPM module
  - **npm install vue** (This method requires you to set up all configuration manually)

We will look at using the CDN as well as using the CLI

Vue File Structure

A Vue.js file is separated into three parts:

- The Template: Where the HTML goes,
- The Script: Where all the logic goes
- The Style: Where all the CSS goes.

```
<template>
  <div id="app">
    <Skills msg='Hello' />
  </div>
</template>

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

export default {
  name: 'app',
  components: {
    Skills
  }
}
</script>

<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Vue.js Data

When a Vue instance is created, it adds all the properties found in its data object to Vue's **reactivity system**. When the values of those properties change, the view will "react", updating to match the new values.

Instance Lifecycle Hooks

Each Vue instance goes through a series of initialization steps when it's created - for example, it needs to set up data observation, compile the template, mount the instance to the DOM, and update the DOM when data changes.

Along the way, it also runs functions called **lifecycle hooks**, giving users the opportunity to add their own code at specific stages.

For example, the **created** hook can be used to run code after an instance is created:

Instance Lifecycle Hooks - beforeCreate

The **beforeCreate** hook runs at the very initialization of your component. data has not been made reactive, and events have not been set up yet.

Instance Lifecycle Hooks - created

In the **created** hook, you will be able to access reactive data and events are active. Templates and Virtual DOM have not yet been mounted or rendered.

Instance Lifecycle Hooks - beforeMount

The **beforeMount** hook runs right before the initial render happens and after the template or render functions have been compiled. Most likely you'll never need to use this hook. Remember, it doesn't get called when doing server-side rendering.

Instance Lifecycle Hooks - mounted

In the mounted hook, you will have full access to the reactive component, templates, and rendered DOM (via. **this.$el**). Mounted is the most-often used lifecycle hook. The most frequently used patterns are fetching data for your component (use created for this instead,) and modifying the DOM, often to integrate non-Vue libraries.

Instance Lifecycle Hooks - beforeUpdate

The **beforeUpdate** hook runs after data changes on your component and the update cycle begins, right before the DOM is patched and re-rendered. It allows you to get the new state of any reactive data on your component before it actually gets rendered.

Instance Lifecycle Hooks - updated

The **updated** hook runs after data changes on your component and the DOM re-renders. If you need to access the DOM after a property change, here is probably the safest place to do it.

Instance Lifecycle Hooks - beforeDestroy

beforeDestroy is fired right before teardown. Your component will still be fully present and functional. If you need to cleanup events or reactive subscriptions, **beforeDestroy** would probably be the time to do it.

Instance Lifecycle Hooks - destroyed

By the time you reach the **destroyed** hook, there's pretty much nothing left on your component.

Everything that was attached to it has been destroyed. You might use the **destroyedhook** to do any last-minute cleanup or inform a remote server that the component was destroyed.

Templating and Interpolation

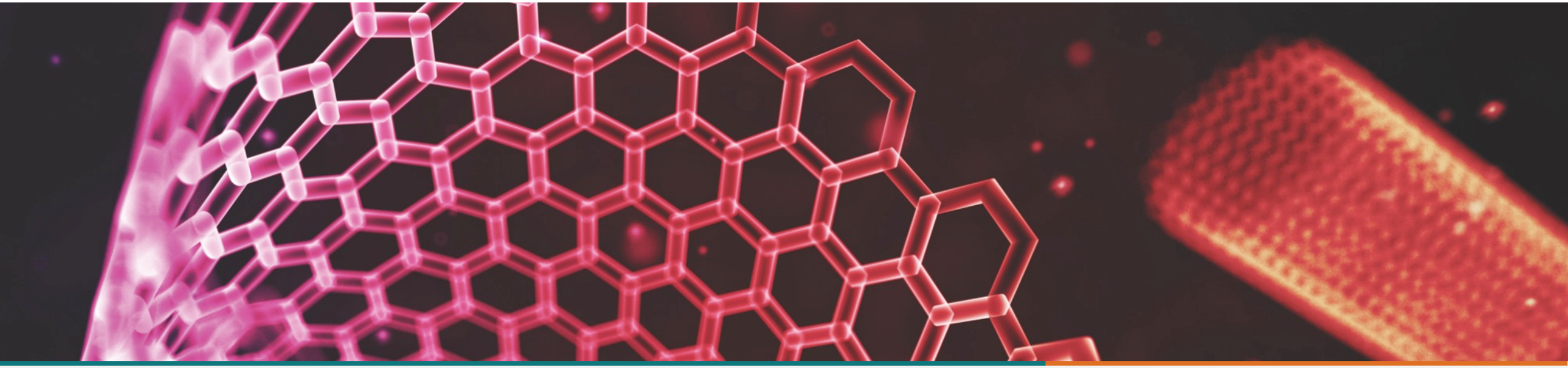
Take the following template snippet:

```
<template>
  <div class="skills">
    {{msg + ", " +  name}}
    <br />
    {{btnState ? 'The button is disabled' : 'The button is active'}}
    <button v-on:click="changeName" v-bind:disabled="btnState">Change Name </button>
  </div>
</template>
```

```
<script>
export default {
  name: 'Skills',
  data(){
    return {
      name:"Patrick",
      btnState: true
    }
  },
```

Setting the initial value for btnState

Directives



What Are Directives?

Directives, are basically like HTML attributes which are added inside templates. They all start with **v-**, to indicate that's a Vue special attribute.

Directives

V-TEXT - Instead of using interpolation, you can use the **v-text** directive. It performs the same job: `<span v-text="name"></span>`

V-ONCE – When using interpolation {{name}} , Any time name changes in your component data, Vue is going to update the value represented in the browser unless you use **v-once**:

```
<span v-once>{{ name }}</span>
```

V-HTML - There are cases however where you want to output HTML and make the browser interpret it. You can use the **v-html** directive: `<span v-html="someHtml"></span>`

V-BIND - Interpolation only works in the tag content. You can't use it on attributes. Attributes must use **v-bind**:

```
<a v-bind:href="url">{{ linkText }}</a>
<a :href="url">{{ linkText }}</a>
```

Directives

V-IF - Conditionally insert / remove the element based on the truthy-ness of the binding value.

V-ON – Attaches an event listener to the element. The event type is denoted by the argument.

V-FOR – Use the v-for directive to render a list of items based on an array.

V-BIND - Interpolation only works in the tag content. You can’t use it on attributes. Attributes must use **v-bind**:

Directives

TWO-WAY BINDING USING V-MODEL

v-model lets us bind a form input element for example, and make it change the Vue data property when the user changes the content of the field:


```
data: {
    message: 'Hello Patrick!',
    shows: [],
    selected: null
},
```

```
<select v-model="selected">
    <option disabled value="">Choose a fruit</option>
    <option>Apple</option>
    <option>Banana</option>
    <option>Strawberry</option>
</select>

<span>Fruit chosen: {{ selected }}</span>
```

Conditional Rendering: v-if, v-else-if, v-else

```
<span v-if="btnState">
  The button is disabled
</span>
<span v-else>
  The button is active
</span>
```

```
<div v-if="Math.random() > 0.5">
  Now you see me
</div>
<div v-else>
  Now you don't
</div>
```

```
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

Looping Through an Array or Object: v-for

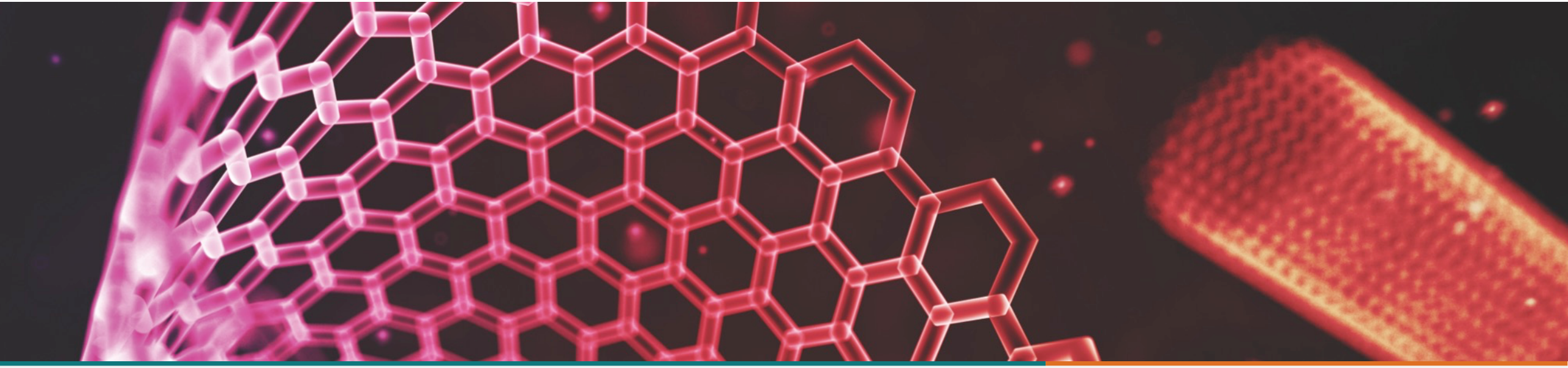
HTML

```
<ul id="example-1">
  <li v-for="item in items">
    {{ item.message }}
  </li>
</ul>
```

JS

```
var example1 = new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ]
  }
})
```

CSS and Styling



CSS in Vue Components

In Vue our CSS is in the style section of the file:

```
<style>
#app {
  font-family: 'Avenir', Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Style in App.vue

CSS - Scoped

We can use scoped in our CSS so that the CSS is for that component only

```
<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
.alert{
  background-color: yellow;
  width: 100%;
  height: 30px;
}
</style>
```

CSS – External CSS File

We can use an external CSS file by setting the src attribute in the style block

```
<!-- Add "scoped" attribute to limit CSS to this component only -->
<style src ="./skills.css" scoped></style>
```

```
ul {
    list-style-type: none;
  }
li {
    font-weight: bold;
}
.alert{
    background-color: yellow;
    width: 100%;
    height: 30px;
  }
```

CSS – Class Binding

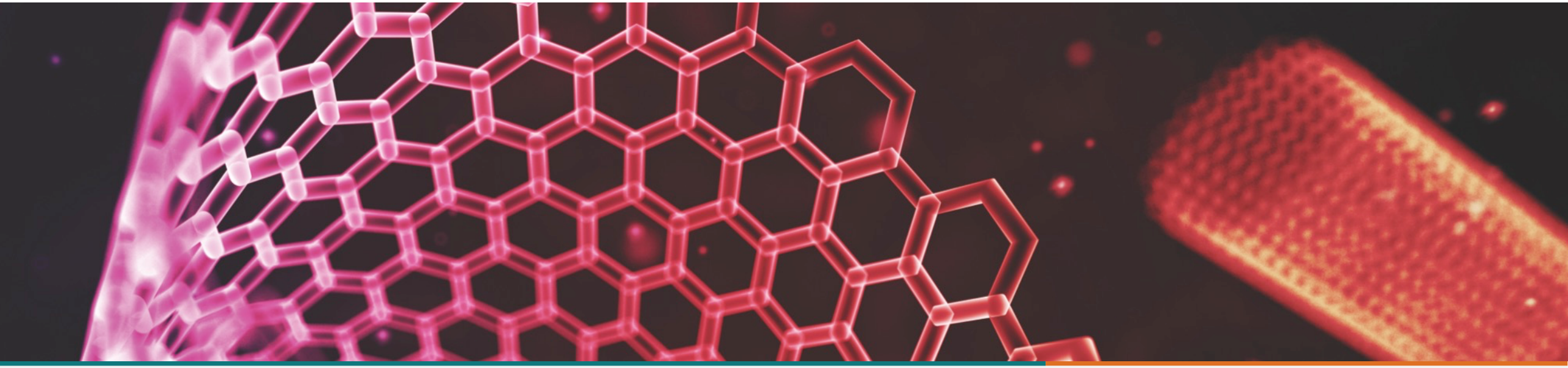
Vue supports class binding

```
<style scoped>
.alert{
  background-color: yellow;
  width: 100%;
  height: 30px;
}
</style>
```

```
<div v-bind:class="{alert: showAlert}">  </div>
```

This will set the class of the div to alert if **showAlert** is **true** otherwise no class will be set

Form Processing and Validation



Form Processing and Validation

In Vue we define inputs much like we do in HTML except we use **v-bind** to bind them to our data variable

```
<input type="text" placeholder="Enter a Skill..." v-model="skill">
```

```
data(){
  return {
    skills:[
      {"skill": "Vue.js"},
      {"skill": "React"},
      {"skill": "Redis"},
    ],
    skill: ''
  }
},
```

Form Processing and Validation

We use a form tag and set the method to call on submit (also prevent the page from reloading on submit)

```
<form @submit.prevent="addSkill">
  <input type="text" placeholder="Enter a Skill..." v-model="skill">
  <p class="alert" v-if="errors.has('skill')"> {{errors.first('skill')}} </p>
</form>
```

Form Processing and Validation

To add the **addSkill** method you would add a methods object to your data like so:

```
<script>
export default {
  name: 'Skills',
  data(){
    return {
      skills:[
        {"skill": "Vue.js"},
        {"skill": "React"},
        {"skill": "Redis"},
      ],
      skill: ''
    }
  },
  methods:{
    addSkill (){
      this.skills.push({skill: this.skill})
      this.skill ='';
    }
  }
}
</script>
```

Form Processing and Validation

For form validation we use the **vee-validate** package. You install it using npm install vee-validate and then modify main.js to import it and include it

```
import Vue from 'vue'
import App from './App.vue'
import VeeValidate from 'vee-validate';
Vue.use(VeeValidate)
Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

Form Processing and Validation

Once we have **vee-validate** installed and imported we can use it in our form. Let's make the skill input so it has to have a min of 5 characters. We also set up a conditional p element to display if there is an error

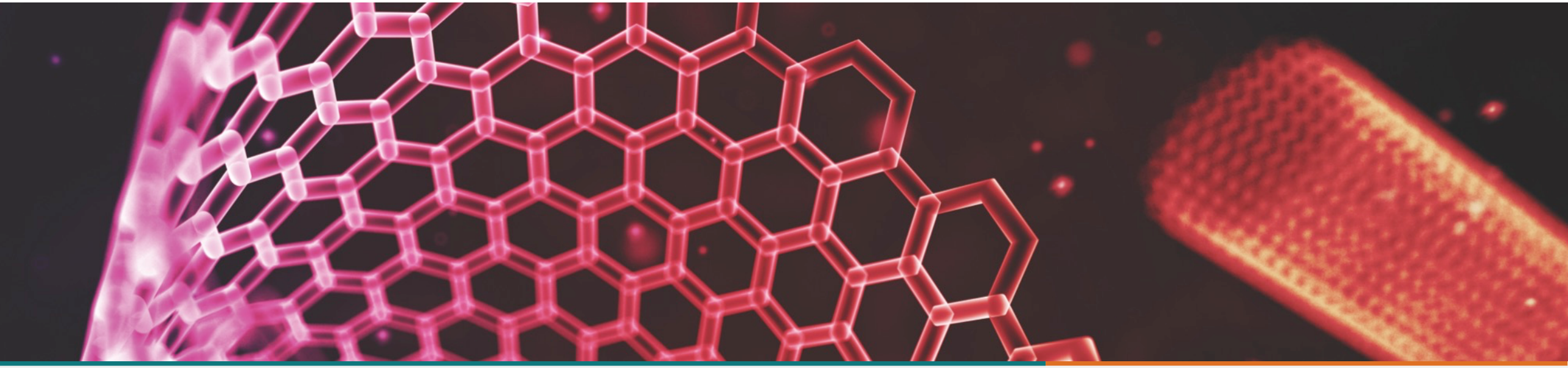
```
<input type="text" placeholder="Enter a Skill..." v-model="skill" v-validate ="'min:5'" name='skill'>
<p class="alert" v-if="errors.has('skill')"> {{errors.first('skill')}} </p>
```

This will show the error if there are less than 5 characters, but it will still allow you to add it to the list. We need to modify the **addSkill** method

Form Processing and Validation

```
methods:{
  addSkill (){
    this.$validator.validateAll().then((result) => {
      if (result){
        this.skills.push({skill: this.skill})
        this.skill ='';
        console.log(`This checkbox value is ${this.checked}`)
      }else{
        console.log ("not valid")
      }
    })
  }
}
```

Animations



Animation

Vue has some cool animation effects. Let's wrap the error p element in a transition component element

```
<transition name ="alert-in">
  <p class="alert" v-if="errors.has('skill')"> {{errors.first('skill')}} </p>
</transition>
```

Once we have that set up, we can write the CSS:

Animation

```
.alert-in-enter-active{
  animation: bounce-in .5s;
}
.alert-in-leave-active{
  animation: bounce-in .5s reverse;
}
@keyframes bounce-in {
  0% {
    transform: scale(0);
  }
  50% {
    transform: scale(1.5);
  }
  100% {
    transform: scale(1);
  }
}
```

Animation Library

We can use an animation library to make our lives easier. We will use Animate.css

```
<transition name ="alert-in" enter-active-class="animated flipInX" leave-active-class="animated flipOutX">
  <p class="alert" v-if="errors.has('skill')"> {{errors.first('skill')}} </p>
</transition>
```

And then import Animate.css (we can do it in the style block)

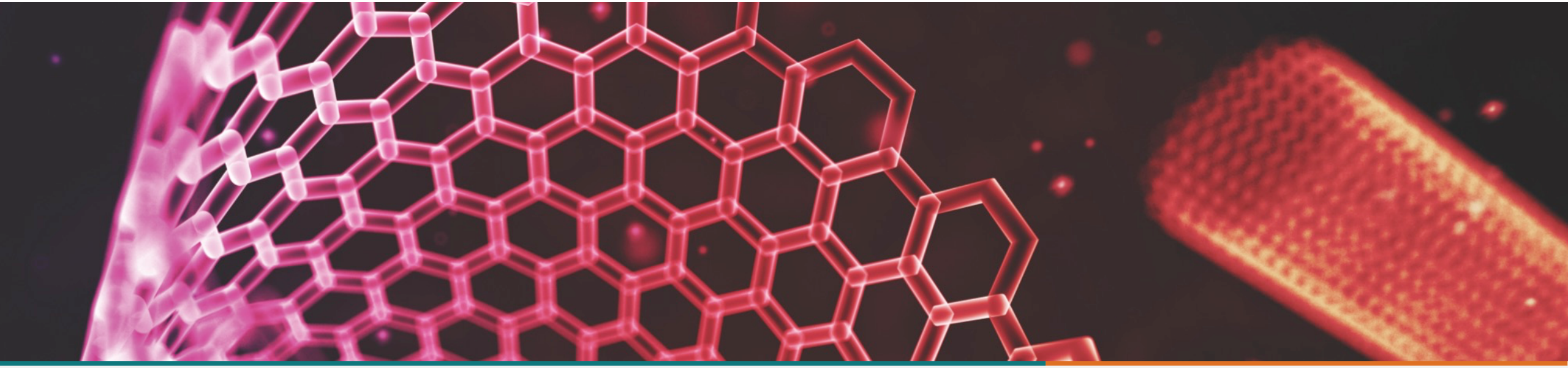
```
<style scoped>
@import "https://cdn.jsdelivr.net/npm/animate.css@3.5.1";

.holder {
    background: #fff;
```

Animation in a List

We can animate a list by using transition group.

```
<ul>
  <transition-group name="list" enter-active-class="animated bounceInUp" leave-active-class="animated bounceOutDown">
    <li v-for="(item,index) in skills" :key='index'>{{item.skill}}</li>
  </transition-group>
</ul>
```

Routing



Routing

We can use Vue Router for routing by installing the vue router.

- npm install vue-router

we then set up our router.js file

```
import Vue from 'vue'
import Router from 'vue-router'
import Skills from './components/Skills.vue'
import About from './components/About.vue'

Vue.use(Router)

export default new Router({
  routes: [
    {
      path: '/',
      name: 'skills',
      component: Skills
    },
    {
      path: '/about',
      name: 'about',
      component: About
    }
  ]
})
```

Routing

We also have to modify main.js

```
import Vue from 'vue'
import App from './App.vue'
import VeeValidate from 'vee-validate';
import router from 'vue-router'
Vue.use(VeeValidate)
Vue.config.productionTip = false

new Vue({
  router,
  render: h => h(App),
}).$mount('#app')
```

Routing

We can then add a nav component to App.vue

```
<nav>
  <router-link to="/">Home</router-link>
  <router-link to="/about">About</router-link>
</nav>
<router-view />
```

We need **<router-view />** so the router templates are displayed

Questions?